The SGRUD Thesis

SGRUD is Growing Rapidly Until Distinction.

Philip Schildkamp

Abstract

Abstract.

Contents

List of Figures

List of Tables

Preformatted

```
*Text*
```

Table

row	row	row
col	col	col
col	col	col
col	col	col

References

Appendix

Module: bin

bin

• **bin**: `Object`

`@sgrud/bin` - The SGRUD CLI.

```
Description
  @sgrud/bin - The SGRUD CLI

Usage
  $ sgrud <command> [options]

Available Commands
  construct    Builds a SGRUD-based project using `microbundle`
  kickstart    Kickstarts a SGRUD-based project using `simple-git`
  postbuild    Replicates exported package metadata for SGRUD-based projects
  runtimify    Creates ESM or UMD bundles for ES6 modules using `microbundle`
  universal    Runs SGRUD in universal (SSR) mode using `puppeteer`

For more info, run any command with the `--help` flag
  $ sgrud construct --help
  $ sgrud kickstart --help

Options
  -v, --version    Displays current version
  -h, --help       Displays this message
```

Defined in

packages/bin/index.ts:1

---

bin.construct

construct

▸ **construct**(options?): `Promise<void>`

constructs a SGRUD-based project using microbundle.

```
Description
  Constructs a SGRUD-based project using `microbundle`

Usage
  $ sgrud construct [...modules] [options]

Options
  --compress    Compress/minify build output  (default true)
  --format      Build specified formats  (default commonjs,modern,umd)
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud construct # Run with default options
```

```
  $ sgrud construct ./project/module # Build ./project/module
  $ sgrud construct ./module --format umd # Build ./module as umd
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.construct();
```

Example

construct `./project/module`:

```
require('@sgrud/bin');

sgrud.bin.construct({
  modules: ['./project/module']
});
```

Example

construct `./module` as `umd`:

```
require('@sgrud/bin');

sgrud.bin.construct({
  modules: ['./module'],
  format: 'umd'
});
```

Parameters

Name	Type	Description
`options`	`Object`	The `options` object.
`options.compress?`	`boolean`	Compress/minify **construct** output. **`Default Value`** `true`
`options.format?`	`string`	**construct** specified formats. **`Default Value`** `'commonjs,modern,umd'`
`options.modules?`	`string[]`	Modules to **construct**. **`Default Value`** `package.json#sgrud.construct`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

An execution Promise.

Defined in packages/bin/src/construct.ts:73

---

bin.kickstart

kickstart

▸ **kickstart**(`options?`): `Promise<void>`

kickstarts a SGRUD-based project using simple-git.

```
Description
  Kickstarts a SGRUD-based project using `simple-git`

Usage
  $ sgrud kickstart [library] [options]

Options
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message
```

```
Examples
  $ sgrud kickstart # Run with default options
  $ sgrud kickstart preact --prefix ./module # Kickstart preact in ./module
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.kickstart();
```

Example

kickstart `preact` in `./module`:

```
require('@sgrud/bin');

sgrud.bin.kickstart({
  prefix: './module',
  library: 'preact'
});
```

Parameters

Name	Type	Description
`options`	`Object`	The `options` object.
`options.library?`	`string`	Library which to base upon. **`Default Value`** `'sgrud'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

An execution Promise.

Defined in packages/bin/src/kickstart.ts:55

---

bin.postbuild

postbuild

▸ **postbuild**(`options?`): `Promise<void>`

Replicates exported package metadata for SGRUD-based projects.

```
Description
  Replicates exported package metadata for SGRUD-based projects

Usage
  $ sgrud postbuild [...modules] [options]

Options
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud postbuild # Run with default options
  $ sgrud postbuild ./project/module # Postbuild ./project/module
  $ sgrud postbuild --prefix ./module # Run in ./module
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.postbuild();
```

Example

postbuild `./project/module`:

```
require('@sgrud/bin');

sgrud.bin.postbuild({
  modules: ['./project/module']
});
```

Example

Run in `./module`:

```
require('@sgrud/bin');

sgrud.bin.postbuild({
  prefix: './module'
});
```

Parameters

Name	Type	Description
`options`	`Object`	The `options` object.
`options.modules?`	`string[]`	Modules to **postbuild**. **Default Value** `package.json#sgrud.postbuild`
`options.prefix?`	`string`	Use an alternative working directory. **Default Value** `'./'`

Returns `Promise<void>`

An execution Promise.

Defined in packages/bin/src/postbuild.ts:67

---

bin.runtimify

runtimify

▸ **runtimify**(`options?`): `Promise<void>`

Creates ESM or UMD bundles for node modules using microbundle.

```
Description
  Creates ESM or UMD bundles for node modules using `microbundle`

Usage
  $ sgrud runtimify [...modules] [options]

Options
  --format      Runtimify bundle format (umd or esm)  (default umd)
  --output      Output file in module root  (default runtimify.[format].js)
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud runtimify # Run with default options
  $ sgrud runtimify @microsoft/fast # Runtimify `@microsoft/fast`
```

Example

Run with default options (not recommended):

```
require('@sgrud/bin');

sgrud.bin.runtimify();
```

Example

runtimify `@microsoft/fast`:

```
require('@sgrud/bin');

sgrud.bin.runtimify({
  modules: ['@microsoft/fast']
});
```

Parameters

Name	Type	Description
`options`	`Object`	The `options` object.
`options.format?`	`string`	**runtimify** bundle format (umd or esm). **`Default Value`** `'umd'`
`options.modules?`	`string[]`	Modules to **runtimify**. **`Default Value`** `package.json#sgrud.runtimify`
`options.output?`	`string`	Output file in module root. **`Default Value`** `'runtimify.[format].js'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

An execution Promise.

Defined in packages/bin/src/runtimify.ts:60

---

bin.universal

universal

▸ **universal**(`options?`): `Promise<void>`

Runs SGRUD in **universal** (SSR) mode using puppeteer.

```
Description
  Runs SGRUD in universal (SSR) mode using `puppeteer`

Usage
  $ sgrud universal [entry] [options]

Options
  --chrome      Chrome executable path  (default /usr/bin/chromium-browser)
  --prefix      Use an alternative working directory  (default ./)
  -H, --host    Host/IP to bind to  (default 127.0.0.1)
  -p, --port    Port to bind to  (default 4000)
  -h, --help    Displays this message

Examples
  $ sgrud universal # Run with default options
  $ sgrud universal --host 0.0.0.0 # Listen on all IPs
  $ sgrud universal -H 192.168.0.10 -p 4040 # Listen on 192.168.0.10:4040
```

`Example`

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.universal();
```

`Example`

Listen on all IPs:

```
require('@sgrud/bin');

sgrud.bin.universal({
  host: '0.0.0.0'
});
```

Example

Listen on `192.168.0.10:4040`:

```
require('@sgrud/bin');

sgrud.bin.universal({
  host: '192.168.0.10',
  port: '4040'
});
```

Parameters

Name	Type	Description
`options`	`Object`	The `options` object.
`options.chrome?`	`string`	Chrome executable path. **`Default Value`** `'/usr/bin/chromium-browser'`
`options.entry?`	`string`	HTML document (relative to `prefix`). **`Default Value`** `'index.html'`
`options.host?`	`string`	Host/IP to bind to. **`Default Value`** `'127.0.0.1'`
`options.port?`	`string`	Port to bind to. **`Default Value`** `'4000'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

An execution Promise.

Defined in packages/bin/src/universal.ts:74

Module: bus

bus

• **bus**: `Object`

`@sgrud/bus` - The SGRUD Software Bus.

The functions and classes found within the **`@sgrud/bus`** module are intended to ease the internal and external real-time communication of applications building upon the SGRUD client libraries. By establishing a Bus between different modules of an application or between the core of an application and plugins extending it, or even between different applications, loose coupling and data transferral can be achieved.

The **`@sgrud/bus`** module includes a standalone JavaScript bundle which is used to Spawn a background Thread upon import of this module. This background Thread is henceforth used as central hub for data exchange. Depending on the runtime environment, either a `new Worker()` or a `new require('worker_threads').Worker()` NodeJS equivalent will be Spawned.

Defined in packages/bus/index.ts:1

---

bus.Bus

Bus

• **Bus**<`I`, `O`>: `Object`

The **Bus** class presents an easy way to establish duplex streams. Through the on-construction supplied Handle the mount point of the created duplex streaming **Bus** within the hierarchical structure of streams handled by the BusHandler is designated. Thereby, all Values emitted by the created **Bus** originate from streams beneath the supplied Handle and when invoking the next method of the implemented Observer contract, the resulting Value will originate from this supplied Handle.

An instantiated **Bus** allows for two modes of observation to facilitate simple and complex use cases. The implemented Subscribable contract allows for observation of the dematerialized Values, while the well-known `Symbol.observable` method provides a way to observe the raw Values, including their originating Handles.

`Example`

Using a duplex streaming **Bus**:

```
import { Bus } from '@sgrud/bus';

const bus = new Bus<string, string>('io.github.sgrud.example');

bus.subscribe({ next: console.log });
bus.next('value');
bus.complete();
```

Type parameters

Name	Description
`I`	The input value type of a **Bus** instance.
`O`	The output value type of a **Bus** instance.

Defined in packages/bus/src/bus/bus.ts:14

packages/bus/src/bus/bus.ts:109

---

bus.Bus.[observable]

[observable][]

▸ **[observable]**(): `Subscribable<Value<O>>`

Well-known `Symbol.observable` method returning a Subscribable. The returned Subscribable emits the raw Values observed by this Bus. By comparison, the implemented subscribe method of the Subscribable interface dematerializes these raw Values before passing them through to the Observer.

`Example`

Subscribe to a raw Bus:

```
import { Bus } from '@sgrud/bus';
import { from } from 'rxjs';

const bus = new Bus<string, string>('io.github.sgrud.example');
from(bus).subscribe(console.log);
```

Returns `Subscribable<Value<O>>`

A Subscribable emitting raw Values.

Defined in packages/bus/src/bus/bus.ts:177

---

bus.Bus.complete

complete

▸ **complete**(): `void`

Implemented **complete** method of the Observer contract. Invoking this method will mark the publishing side of this duplex Bus as **complete**d.

`Example`

complete a Bus:

```ts
import { Bus } from '@sgrud/bus';

const bus = new Bus<string, string>('io.github.sgrud.example');
bus.complete();
```

Returns `void`

Implementation of Observer.complete

Defined in packages/bus/src/bus/bus.ts:195

---

bus.Bus.constructor

constructor

• **new Bus**<`I`, `O`>(`handle`)

Public Bus **constructor**. The Handle supplied to this **constructor** is assigned as `readonly` on the constructed Bus instance and will be used to determine the mount point of this duplex stream within the hierarchical structure of streams handled by the BusHandler.

Type parameters

Name
`I`
`O`

Parameters

Name	Type	Description
`handle`	'${string}.${string}.${string}'	The Handle to publish this Bus under.

Defined in packages/bus/src/bus/bus.ts:134

---

bus.Bus.error

error

▸ **error**(`error`): `void`

Implemented **error** method of the Observer contract. Invoking this method will throw the supplied `error` on the publishing side of this duplex Bus.

`Example`

Throw an **error** through a Bus:

```ts
import { Bus } from '@sgrud/bus';

const bus = new Bus<string, string>('io.github.sgrud.example');
bus.error(new Error('example'));
```

Parameters

Name	Type	Description
`error`	`unknown`	The `error` to publish.

Returns `void`

Implementation of Observer.error

Defined in packages/bus/src/bus/bus.ts:215

---

bus.Bus.handle

handle

• `Readonly` **handle**: '${string}.${string}.${string}'

The Handle to publish this Bus under.

Defined in packages/bus/src/bus/bus.ts:139

---

bus.Bus.next

next

▸ **next**(`value`): `void`

Implemented **next** method of the Observer contract. Invoking this method will provide any observer of the publishing side of this duplex Bus with the **next** `value`.

`Example`

Supplying a Bus with a **next** value:

```
import { Bus } from '@sgrud/bus';

const bus = new Bus<string, string>('io.github.sgrud.example');
bus.next('value');
```

Parameters

Name	Type	Description
`value`	`I`	The **next** `value` to publish.

Returns `void`

Implementation of Observer.next

Defined in packages/bus/src/bus/bus.ts:235

---

bus.Bus.subscribe

subscribe

▸ **subscribe**(`observer?`): `Unsubscribable`

Implemented **subscribe** method of the Subscribable contract. Invoking this method while supplying an `observer` will **subscribe** the supplied `observer` to any changes on the observed side of this duplex Bus.

`Example`

subscribe to a dematerialized Bus:

```
import { Bus } from '@sgrud/bus';

const bus = new Bus<string, string>('io.github.sgrud.example');
bus.subscribe({ next: console.log });
```

Parameters

Name	Type	Description
`observer?`	`Partial<Observer<O>>`	The `observer` to **subscribe** to this Bus.

Returns `Unsubscribable`

An Unsubscribable of the ongoing observation.

Implementation of Subscribable.subscribe

Defined in packages/bus/src/bus/bus.ts:257

---

bus.Bus.observe

observe

• `Private` `Readonly` **observe**: `Observable<Value<O>>`

The **observe**d side of this Bus. The Observable assigned to this property is used to fullfil the Subscribable contract and is obtained through the BusHandler.

Defined in packages/bus/src/bus/bus.ts:116

---

bus.Bus.publish

publish

• `Private` `Readonly` **publish**: `Subject<I>`

The **publish**ing side of this Bus. The Subject assigned to this property is used to fullfil the Observer contract and is provided to the BusHandler for **publish**ment.

Defined in packages/bus/src/bus/bus.ts:123

---

bus.Bus

Bus

• **Bus**: `Object`

The **Bus** namespace contains types and interfaces used and intended to be used in conjunction with the Singleton BusHandler class. This namespace contains the Handle string literal type helper, designating the hierarchical mount-point of any **Bus**, as well as the Value type helper, describing the data and state a **Bus** may transport.

See

Bus

Defined in packages/bus/src/bus/bus.ts:14

packages/bus/src/bus/bus.ts:109

---

bus.Bus.Handle

Handle

T **Handle**: '${string}.${string}.${string}'

The **Handle** string literal helper type enforces any assigned value to contain at least three dots. It represents a type constraint which should be thought of as domain name in reverse notation. All employed **Handle**s thereby designate a hierarchical structure, which the BusHandler in conjunction with the BusWorker operate upon.

Example

Library-wide **Handle**:

```ts
import { type Bus } from '@sgrud/bus';

const busHandle: Bus.Handle = 'io.github.sgrud';
```

Example

An invalid **Handle**:

```ts
import { type Bus } from '@sgrud/bus';

const busHandle: Bus.Handle = 'org.example';
// Type [...] is not assignable to type 'Handle'.
```

See

BusHandler

Defined in

packages/bus/src/bus/bus.ts:42

---

bus.Bus.Value

Value

T **Value**<T>: `ObservableNotification<T>` & { `handle`: `Handle` }

The **Value** type helper extends the ObservableNotification type and describes the shape of all values emitted by any stream handled by the BusHandler. As those streams are Observables, which are dynamically combined through their hierarchical structure denoted by their corresponding Handles and therefore may emit from more than one Handle, each **Value** emitted by any bus contains its originating Handle.

Example

Logging emitted **Value**s:

```ts
import { BusHandler } from '@sgrud/bus';

const busHandler = new BusHandler();
busHandler.observe('io.github.sgrud').subscribe(console.log);
// { handle: 'io.github.sgrud.example', type: 'N', value: 'published' }
```

See

BusHandler

Type parameters

Name	Description
T	The Bus **Value** type.

Defined in

packages/bus/src/bus/bus.ts:67

---

bus.BusHandler

BusHandler

• **BusHandler**: `Object`

The **BusHandler** implements and orchestrates the establishment, transferral and deconstruction of any number of Observable streams. It operates in conjunction with the BusWorker Thread which is run in the background. To designate and organize different Observable streams, the string literal helper type Handle is employed. As an example, let the following hierarchical structure be given:

```
io.github.sgrud
├── io.github.sgrud.core
│   ├── io.github.sgrud.core.kernel
│   └── io.github.sgrud.core.transit
├── io.github.sgrud.data
│   ├── io.github.sgrud.data.model.current
│   └── io.github.sgrud.data.model.global
└── io.github.sgrud.shell
│   └── io.github.sgrud.shell.route
└── io.github.sgrud.store
    ├── io.github.sgrud.store.global
    └── io.github.sgrud.store.local
```

Depending on the Handle, one may observe all established streams beneath the root `io.github.sgrud` Handle or only one specific stream, e.g., `io.github.sgrud.core.kernel`. The Observable returned from the observe method will emit all Values originating from all streams beneath the root Handle in the first case, or only Values from one stream, in the second case.

`Decorator`

Singleton

`See`

BusWorker

Defined in

packages/bus/src/handler/handler.ts:46

---

bus.BusHandler.[observable]

[observable][]

▸ `Static` **[observable]**(): `Subscribable<BusHandler>`

Static `Symbol.observable` method returning a Subscribable. The returned Subscribable mirrors the private loader and is used for initializations after the BusHandler has been successfully initialized.

`Example`

Subscribe to the BusHandler:

```
import { BusHandler } from '@sgrud/bus';
import { from } from 'rxjs';

from(BusHandler).subscribe(console.log);
```

Returns

`Subscribable<BusHandler>`

A Subscribable emitting this BusHandler.

Defined in

packages/bus/src/handler/handler.ts:72

---

bus.BusHandler.loader

loader

■ `Static` `Private` **loader**: `ReplaySubject<BusHandler>`

Private static ReplaySubject used as the BusHandler **loader**. This **loader** emits once after the BusHandler has been successfully initialized.

Defined in packages/bus/src/handler/handler.ts:53

---

bus.BusHandler.constructor

constructor

• **new BusHandler**(`source?`)

Public BusHandler **constructor**. As the BusHandler is a Singleton class, this **constructor** is only invoked the first time it is targeted by the `new` operator. Upon this first invocation, the worker property is assigned an instance of the BusWorker Thread while using the supplied `source`, if any.

`Throws`

A ReferenceError when the environment is incompatible.

Parameters

Name	Type	Description
`source?`	`string`	An optional Module `source`.

Defined in packages/bus/src/handler/handler.ts:104

---

bus.BusHandler.observe

observe

▸ **observe**<`T`>(`handle`): `Observable<Value<T>>`

Invoking this method **observe**s the Observable stream represented by the supplied `handle`. The method will return an Observable originating from the BusWorker which emits all Values published under the supplied `handle`. When the **observe** method is invoked with `'io.github.sgrud'`, all streams hierarchically beneath this Handle, e.g., `'io.github.bus.status'`, will also be emitted by the returned Observable.

`Example`

observe the `'io.github.sgrud'` stream:

```
import { BusHandler } from '@sgrud/bus';

const busHandler = new BusHandler();
const handle = 'io.github.sgrud.example';

busHandler.observe(handle).subscribe(console.log);
```

Type parameters

Name	Description
`T`	The type of the **observe**d Observable stream.

Parameters

Name	Type	Description
`handle`	‘${string}.${string}.${string}‘	The Handle to **observe**.

Returns `Observable<Value<T>>`

An Observable bus for `handle`.

Defined in packages/bus/src/handler/handler.ts:160

---

bus.BusHandler.publish

publish

▸ **publish**<T>(handle, stream): Observable<void>

Invoking this method **publish**es the supplied Observable `stream` under the supplied `handle`. This method returns an Observable of the **publish**ment of the supplied Observable `stream` under the supplied `handle` with the BusWorker. When the **publish**ed `source` Observable completes, the registration within the BusWorker will automatically self-destruct.

Example

publish a stream under `'io.github.sgrud.example'`:

```
import { BusHandler } from '@sgrud/bus';
import { of } from 'rxjs';

const busHandler = new BusHandler();
const handle = 'io.github.sgrud.example';
const stream = of('published');

busHandler.publish(handle, stream).subscribe();
```

Type parameters

Name	Description
T	The type of the **publish**ed Observable stream.

Parameters

Name	Type	Description
handle	`${string}.${string}.${string}`	The Handle to **publish** under.
stream	ObservableInput<T>	The Observable `stream` for `handle`.

Returns Observable<void>

An Observable of the `stream` **publish**ment.

Defined in packages/bus/src/handler/handler.ts:193

---

bus.BusHandler.uplink

uplink

▸ **uplink**(handle, url): Observable<Subscription>

Invoking this method **uplink**s the supplied `handle` to the supplied `url` by establishing a WebSocket connection between the endpoint behind the supplied `url` and the BusWorker. This method returns an Observable of the **uplink** Subscription which can be used to cancel the **uplink**. When the **uplink**ed WebSocket is closed or throws an error, it is automatically cleaned up and unsubscribed from.

Example

uplink the `'io.github.sgrud.uplink'` Handle:

```
import { BusHandler } from '@sgrud/bus';

const busHandler = new BusHandler();
const handle = 'io.github.sgrud.example';
const url = 'https://example.com/websocket';

const uplink = busHandler.uplink(handle, url).subscribe();
```

Parameters

Name	Type	Description
`handle`	'${string}.${string}.${string}'	The Handle to **uplink**.
`url`	`string`	The endpoint `url` to establish an **uplink** to.

Returns `Observable<Subscription>`

An Observable of the **uplink** Subscription.

Defined in packages/bus/src/handler/handler.ts:231

---

bus.BusHandler.worker

worker

• `Readonly` **worker**: `Thread<BusWorker>`

The **worker** Thread and main background workhorse. The underlying BusWorker is run inside a Worker context in the background and transparently handles published and observed streams and the aggregation of their values depending on their Handle, i.e., hierarchy.

See

BusWorker

Defined in packages/bus/src/handler/handler.ts:92

---

bus.BusQuerier

BusQuerier

• **BusQuerier**: `Object`

The **BusQuerier** implements an Bus based Querier, i.e., extension of the abstract Querier base class, allowing Model queries to be executed via a Bus. To use this class, provide it to the Linker by either extending it, and decorating the extending class with the Target decorator, or by preemptively supplying an instance of this class to the Linker.

Example

Provide the **BusQuerier** to the Linker:

```
import { BusQuerier } from '@sgrud/bus';
import { Linker } from '@sgrud/core';

new Linker<typeof BusQuerier>([
  [BusQuerier, new BusQuerier('io.github.sgrud.example')]
]);
```

See

- Model
- Querier

Defined in packages/bus/src/bus/querier.ts:28

---

bus.BusQuerier.[provide]

[provide][]

■ `Static` `Readonly` **[provide]**: `"sgrud.data.Querier"`

Magic string by which this class is provided.

See

provide

Inherited from Querier.[provide]

Defined in packages/data/src/querier/querier.ts:96

---

bus.BusQuerier.commit

commit

▸ **commit**(`operation`, `variables`): `Observable<unknown>`

Overridden **commit** method of the Querier base class. When this Querier is made available via the Linker, this overridden **commit** method is called when this Querier claims the highest priority to **commit** an Operation, depending on the Model from which the Operation originates.

Parameters

Name	Type	Description
`operation`	`subscription ${string}` | `mutation ${string}` | `query ${string}`	The Operation to be **commit**ted.
`variables`	`Variables`	Any Variables within the `operation`.

Returns `Observable<unknown>`

An Observable of the **commit**ted `operation`.

Overrides Querier.commit

Defined in packages/bus/src/bus/querier.ts:82

---

bus.BusQuerier.constructor

constructor

• **new BusQuerier**(`handle`, `prioritize?`)

Public **constructor** consuming the `handle` Model queries should be committed through, and an dynamic or static `prioritize` value. The `prioritize` value may either be a mapping of Models to corresponding priorities or a static priority for this Querier.

Parameters

Name	Type	Default value	Description
`handle`	`${string}.${string}.${string}`	`undefined`	The Handle to commit queries under.
`prioritize`	`number` | `Map<Type<Model<any>>, number>`	`0`	The dynamic or static prioritization.

Overrides Querier.constructor

Defined in packages/bus/src/bus/querier.ts:51

---

bus.BusQuerier.priority

priority

▸ **priority**(`model`): `number`

Overridden **priority** method of the Querier base class. When an Operation is to be committed, this method is called with the respective `model` Type and returns the claimed **priority** to commit this Model.

Parameters

Name	Type	Description
`model`	`Type<Model<any>>`	The Model to be committed.

Returns `number`

The numeric **priority** of this Querier implementation.

Overrides Querier.priority

Defined in packages/bus/src/bus/querier.ts:107

---

bus.BusQuerier.types

types

• `Readonly` **types**: `Set<Type>`

A set containing the Types this BusQuerier handles. As a Bus is a long-lived duplex stream, this Querier can handle `'mutation'`, `'query'` and `'subscription'` **types**.

Overrides Querier.types

Defined in packages/bus/src/bus/querier.ts:36

---

bus.BusQuerier.handle

handle

• `Private` `Readonly` **handle**: ‘${string}.${string}.${string}‘

The Handle to commit queries under.

Defined in packages/bus/src/bus/querier.ts:56

---

bus.BusQuerier.prioritize

prioritize

• `Private` `Readonly` **prioritize**: `number` | `Map<Type<Model<any>>, number>` = `0`

The dynamic or static prioritization.

`Default Value`

`0`

`See`

priority

Defined in packages/bus/src/bus/querier.ts:65

---

bus.BusWorker

BusWorker

• **BusWorker**: `Object`

The **BusWorker** is a background Thread which is Spawned by the BusHandler to handle all published and observed streams, uplinks and their aggregation depending on their hierarchy.

`Decorator`

Thread

`Decorator`

Singleton

`See`

BusHandler

Defined in packages/bus/src/worker/index.ts:24

---

bus.BusWorker.constructor

constructor

• **new BusWorker**()

Public **constructor**. This **constructor** is called once when the BusHandler Spawns this BusWorker.

`Remarks`

This method should only be invoked by the BusHandler.

Defined in packages/bus/src/worker/index.ts:52

---

bus.BusWorker.observe

observe

▸ **observe**<T>(`handle`): `Promise<Observable<Value<T>>>`

Invoking this method **observe**s all Observable streams under the supplied `handle` by mergeing all streams which are published under the supplied `handle`.

`Remarks`

This method should only be invoked by the BusHandler.

Type parameters

Name
T

Parameters

Name	Type	Description
`handle`	‘${string}.${string}.${string}‘	The Handle to **observe**.

Returns `Promise<Observable<Value<T>>>`

An Observable stream for `handle`.

Defined in packages/bus/src/worker/index.ts:68

---

bus.BusWorker.publish

publish

▸ **publish**<T>(`handle`, `stream`): `Promise<void>`

Invoking this method **publish**es the supplied ObservableInput `stream` under the supplied `handle`. Any emittance of the **publish**ed `stream` will be materialized into Values and replayed once to every observer.

`Throws`

A ReferenceError on collision of `handle`s.

`Remarks`

This method should only be invoked by the BusHandler.

Type parameters

Name
T

Parameters

Name	Type	Description
`handle`	`${string}.${string}.${string}`	The Handle to **publish** under.
`stream`	`ObservableInput<T>`	The ObservableInput `stream` for `handle`.

Returns `Promise<void>`

A Promise of the `stream` **publish**ment.

Defined in packages/bus/src/worker/index.ts:111

---

bus.BusWorker.uplink

uplink

▸ **uplink**(`handle`, `url`): `Promise<Subscription>`

Invoking this method **uplink**s the supplied `handle` to the supplied `url` by establishing a WebSocket connection between the endpoint behind the supplied `url` and this BusWorker. It is assumed, that all messages emanating from the WebSocket endpoint conform to the Value type and are therefore treated as such. This treatment includes the filtering of all received and submitted messages by comparing their corresponding Handle and the supplied `handle`.

`Throws`

A ReferenceError on collision of `handle`s.

`Remarks`

This method should only be invoked by the BusHandler.

Parameters

Name	Type	Description
`handle`	`'${string}.${string}.${string}'`	The Handle to **uplink**.
`url`	`string`	The endpoint `url` to establish an **uplink** to.

Returns `Promise<Subscription>`

A Promise of the Subscription to the **uplink**.

Defined in packages/bus/src/worker/index.ts:150

---

bus.BusWorker.changes

changes

• `Private` `Readonly` **changes**: `BehaviorSubject<BusWorker>`

BehaviorSubject emitting every time when **changes** occur on the internal streams or uplinks mappings. This emittance is used to recombine the Observable streams which were previously obtained to through use of the observe method.

Defined in packages/bus/src/worker/index.ts:32

---

bus.BusWorker.streams

streams

• `Private` `Readonly` **streams**: `Map<'${string}.${string}.${string}', Observable<Value<unknown>>>`

Internal Mapping containing all established **streams**. Updating this map should always be accompanied by an emittance of changes.

Defined in packages/bus/src/worker/index.ts:38

---

bus.BusWorker.uplinks

uplinks

• `Private` `Readonly` **uplinks**: `Map<'${string}.${string}.${string}', Observable<Value<unknown>>>`

Internal Mapping containing all established **uplinks**. Updating this map should always be accompanied by an emittance of changes.

Defined in packages/bus/src/worker/index.ts:44

---

bus.Observe

Observe

▸ **Observe**(`handle`, `suffix?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. Applying this decorator replaces the decorated property with a getter returning an Observable stream which **Observe**s all values originating from the supplied `handle`. Depending on the value of the `suffix` parameter, this Observable stream is either assigned directly to the prototype using the supplied `handle`, or, if a truthy value is supplied for the `suffix` parameter, this value is assumed to reference another property of the class containing this decorated property. The first truthy value assigned to this `suffix` property on an instance of the class containing this **Stream** decorator will then be used to suffix the supplied `handle` which is to be **Observe**d and assign the resulting Observable stream to the decorated instance property.

This decorator is more or less the opposite of the Publish decorator, while both rely on the BusHandler to fulfill contracts.

Example

Observe the 'io.github.sgrud.example' stream:

```
import { type Bus, Observe } from '@sgrud/bus';
import { type Observable } from 'rxjs';

export class Observer {

  @Observe('io.github.sgrud.example')
  public readonly stream!: Observable<Bus.Value<unknown>>;

}

Observer.prototype.stream.subscribe(console.log);
```

Example

Observe the 'io.github.sgrud.example' stream:

```
import { type Bus, Observe } from '@sgrud/bus';
import { type Observable } from 'rxjs';

export class Observer {

  @Observe('io.github.sgrud', 'suffix')
  public readonly stream!: Observable<Bus.Value<unknown>>;

  public constructor(
    public readonly suffix: string
  ) { }

}

const observer = new Observer('example');
observer.stream.subscribe(console.log);
```

See

- BusHandler
- Publish
- Stream

Parameters

Name	Type	Description
handle	`${string}.${string}.${string}`	The Handle to **Observe**.
suffix?	PropertyKey	An optional suffix property for the handle.

Returns fn

A prototype property decorator.

▸ (prototype, propertyKey): void

Parameters

Name	Type
prototype	object
propertyKey	PropertyKey

Returns void

Defined in packages/bus/src/handler/observe.ts:66

---

bus.Publish

Publish

▸ **Publish**(`handle`, `suffix?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. This decorator **Publish**es a newly instantiated Subject under the supplied `handle` and assigns it to the decorated property. Depending on the value of the `suffix` parameter, this newly instantiated Subject is either assigned directly to the prototype and **Publish**ed using the supplied `handle`, or, if a truthy value is supplied for the `suffix` parameter, this value is assumed to reference another property of the class containing this decorated property. The first truthy value assigned to this `suffix` property on an instance of the class containing this **Publish** decorator will then be used to suffix the supplied `handle` upon **Publish**ment of the newly instantiated Subject, which is assigned to the decorated instance property.

Through these two different modes of operation, the Subject that will be **Publish**ed can be assigned statically to the prototype of the class containing the decorated property, or this assignment can be deferred until an instance of the class containing the decorated property is constructed and a truthy value is assigned to its `suffix` property.

This decorator is more or less the opposite of the Observe decorator, while both rely on the BusHandler to fulfill contracts. Furthermore, precautions should be taken to ensure the completion of the **Publish**ed Subject as memory leaks may occur due to dangling subscriptions.

Example

Publish the `'io.github.sgrud.example'` stream:

```
import { Publish } from '@sgrud/bus';
import { type Subject } from 'rxjs';

export class Publisher {

  @Publish('io.github.sgrud.example')
  public readonly stream!: Subject<unknown>;

}

Publisher.prototype.stream.next('value');
Publisher.prototype.stream.complete();
```

Example

Publish the `'io.github.sgrud.example'` stream:

```
import { Publish } from '@sgrud/bus';
import { type Subject } from 'rxjs';

export class Publisher {

  @Publish('io.github.sgrud', 'suffix')
  public readonly stream: Subject<unknown>;

  public constructor(
    private readonly suffix: string
  ) {}

}

const publisher = new Publisher('example');
publisher.stream.next('value');
publisher.stream.complete();
```

See

- BusHandler
- Observe
- Stream

Parameters

Name	Type	Description
`handle`	`${string}.${string}.${string}`	The Handle to **Publish**.
`suffix?`	`PropertyKey`	An optional `suffix` property for the `handle`.

Returns `fn`

A prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/bus/src/handler/publish.ts:76

---

bus.Stream

Stream

▸ **Stream**(`handle`, `suffix?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. Applying this decorator replaces the decorated property with a getter returning a Bus, thereby allowing to duplex **Stream** values designated by the supplied `handle`. Depending on the value of the `suffix` parameter, this Bus is either assigned directly to the prototype using the supplied `handle`, or, if a truthy value is supplied for the `suffix` parameter, this value is assumed to reference another property of the class containing this decorated property. The first truthy value assigned to this `suffix` property on an instance of the class containing this **Stream** decorator will then be used to suffix the supplied `handle` upon construction of the Bus, which is assigned to the decorated instance property.

Through these two different modes of operation, a Bus can be assigned statically to the prototype of the class containing the decorated property, or this assignment can be deferred until an instance of the class containing the decorated property is constructed and a truthy value is assigned to its `suffix` property.

Example

Stream `'io.github.sgrud.example'`:

```
import { type Bus, Stream } from '@sgrud/bus';

export class Streamer {

  @Stream('io.github.sgrud.example')
  public readonly stream!: Bus<unknown, unknown>;

}

Streamer.prototype.stream.next('value');
Streamer.prototype.stream.complete();

Streamer.prototype.stream.subscribe({
  next: console.log
});
```

Example

Stream `'io.github.sgrud.example'`:

```
import { type Bus, Stream } from '@sgrud/bus';

export class Streamer {

  @Stream('io.github.sgrud', 'suffix')
  public readonly stream!: Bus<unknown>;

  public constructor(
    public readonly suffix: string
  ) { }
```

```
}

const streamer = new Streamer('example');
streamer.stream.next('value');
streamer.stream.complete();

streamer.stream.subscribe({
  next: console.log
});
```

See

- BusHandler
- Observe
- Publish

Parameters

Name	Type	Description
`handle`	`'${string}.${string}.${string}'`	The Handle to **Stream**.
`suffix?`	`PropertyKey`	An optional `suffix` property for the `handle`.

Returns `fn`

A prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/bus/src/handler/stream.ts:75

Module: core

core

• **core**: `Object`

`@sgrud/core` - The SGRUD Core Module.

The functions and classes found within the **`@sgrud/core`** module represent the base upon which the SGRUD client libraries are built. Therefore, most of the code provided within this module does not aim at fulfilling one specific high-level need, but is used and intended to be used as low-level building blocks for downstream projects. This practice is employed throughout the SGRUD client libraries, as all modules depend on this core module. By providing the core functionality within this singular module, all downstream SGRUD modules should be considered opt-in functionality which may be used within projects building upon the SGRUD client libraries.

Defined in packages/core/index.ts:1

---

core.Assign

Assign

T **Assign**<S, T>: { [K in keyof (S & T)]: K extends keyof S ? S[K] : K extends keyof T ? T[K] : never }

Type helper **Assign**ing the own property types of all of the enumerable own properties from a source type to a target type.

Example

Assign `valueOf()` to `string`:

```
import { type Assign } from '@sgrud/core';

const str = 'Hello world' as Assign<{
  valueOf(): 'Hello world';
}, string>;
```

Type parameters

Name	Description
S	The source type to **Assign** from.
T	The target type to **Assign** to.

Defined in

packages/core/src/typing/assign.ts:18

---

core.Factor

Factor

▸ **Factor**<K>(targetFactory, transient?): (prototype: object, propertyKey: PropertyKey) => void

Prototype property decorator factory. Applying this decorator replaces the decorated prototype property with a getter, which returns the linked instance of a Targeted constructor, referenced by the `targetFactory`. Depending on the supplied `transient` value, the target constructor is invoked to construct (and link) an instance, if none is linked beforehand.

Example

Factor an eager and lazy service:

```
import { Factor } from '@sgrud/core';
import { EagerService, LazyService } from './services';

export class ServiceHandler {

  @Factor(() => EagerService)
  private readonly service!: EagerService;

  @Factor(() => LazyService, true)
  private readonly service?: LazyService;

}
```

See

- Linker
- Target

Type parameters

Name	Type	Description
K	extends () => any	The Targeted constructor type.

Parameters

Name	Type	Default value	Description
`targetFactory`	() => `K`	`undefined`	A forward reference to the target constructor.
`transient`	`boolean`	`false`	Wether an instance is constructed if none is linked.

Returns `fn`

A prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/core/src/linker/factor.ts:35

---

core.Http

Http

• `Abstract` **Http**: `Object`

The abstract **Http** class is a thin wrapper around the ajax method. The main function of this wrapper is to pipe all requests through a chain of classes extending the abstract Proxy class. Thereby interceptors for various requests can be implemented to, e.g., provide API credentials etc.

See

Proxy

Defined in packages/core/src/http/http.ts:12

packages/core/src/http/http.ts:56

---

core.Http.delete

delete

▸ `Static` **delete**<`T`>(`url`): `Observable`<`Response`<`T`>>

Fires an Http **delete** request against the supplied `url` upon subscription.

Example

Fire an HTTP **delete** request against `https://example.com`:

```
import { Http } from '@sgrud/core';

Http.delete('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
`T`	The Response type.

Parameters

Name	Type	Description
`url`	`string`	The `url` to Http **delete**.

Returns `Observable<Response<T>>`

An Observable of the Response.

Defined in packages/core/src/http/http.ts:74

---

core.Http.get

get

▸ `Static` **get**<T>(`url`): `Observable<Response<T>>`

Fires an Http **get** request against the supplied `url` upon subscription.

`Example`

Fire an HTTP **GET** request against `https://example.com`:

```
import { Http } from '@sgrud/core';

Http.get('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
`T`	The Response type.

Parameters

Name	Type	Description
`url`	`string`	The `url` to Http **get**.

Returns `Observable<Response<T>>`

An Observable of the Response.

Defined in packages/core/src/http/http.ts:94

---

core.Http.head

head

▸ `Static` **head**<T>(`url`): `Observable<Response<T>>`

Fires an Http **head** request against the supplied `url` upon subscription.

`Example`

Fire an HTTP **head** request against `https://example.com`:

```
import { Http } from '@sgrud/core';

Http.head('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
`T`	The Response type.

Parameters

Name	Type	Description
`url`	`string`	The `url` to Http **head**.

Returns `Observable<Response<T>>`

An Observable of the Response.

Defined in packages/core/src/http/http.ts:114

---

core.Http.patch

patch

▸ `Static` **patch**<T>(`url`, `body`): `Observable<Response<T>>`

Fires an Http **patch** request against the supplied `url` containing the supplied `body` upon subscription.

`Example`

Fire an HTTP **patch** request against `https://example.com`:

```
import { Http } from '@sgrud/core';

Http.patch('https://example.com', {
  data: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
`T`	The Response type.

Parameters

Name	Type	Description
`url`	`string`	The `url` to Http **patch**.
`body`	`unknown`	The `body` of the Request.

Returns `Observable<Response<T>>`

An Observable of the Response.

Defined in packages/core/src/http/http.ts:137

---

core.Http.post

post

▸ `Static` **post**<T>(`url`, `body`): `Observable<Response<T>>`

Fires an Http **post** request against the supplied `url` containing the supplied `body` upon subscription.

`Example`

Fire an HTTP **post** request against `https://example.com`:

```
import { Http } from '@sgrud/core';

Http.post('https://example.com', {
  data: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
T	The Response type.

Parameters

Name	Type	Description
`url`	`string`	The `url` to Http **post**.
`body`	`unknown`	The `body` of the Request.

Returns `Observable<Response<T>>`

An Observable of the Response.

Defined in packages/core/src/http/http.ts:163

---

core.Http.put

put

▸ `Static` **put**<T>(`url`, `body`): `Observable<Response<T>>`

Fires an Http **put** request against the supplied `url` containing the supplied `body` upon subscription.

`Example`

Fire an HTTP **put** request against `https://example.com`:

```
import { Http } from '@sgrud/core';

Http.put('https://example.com', {
  data: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
T	The Response type.

Parameters

Name	Type	Description
`url`	`string`	The `url` to Http **put**.
`body`	`unknown`	The `body` of the Request.

Returns `Observable<Response<T>>`

An Observable of the Response.

Defined in packages/core/src/http/http.ts:189

---

core.Http.request

request

▸ `Static` **request**<T>(request): `Observable<Response<T>>`

Fires a custom Request. Use this method for more fine-grained control over the outgoing Request.

`Example`

Fire an HTTP custom request against `https://example.com`:

```
import { Http } from '@sgrud/core';

Http.request({
  method: 'GET',
  url: 'https://example.com',
  headers: { 'x-example': 'value' }
}).subscribe(console.log);
```

Type parameters

Name	Description
T	The Response type.

Parameters

Name	Type	Description
request	AjaxConfig	The Request to be **request**ed.

Returns `Observable<Response<T>>`

An Observable of the Response.

Defined in packages/core/src/http/http.ts:216

---

core.Http.handle

handle

▸ **handle**<T>(request): `Observable<Response<T>>`

Generic **handle** method, enforced by the Handler interface. Main method of the this class. Internally pipes the `request` through all linked classes extending Proxy.

Type parameters

Name	Description
T	The type of the **handle**d Response.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	The Request to be **handle**d.

Returns `Observable<Response<T>>`

An Observable of the Response.

Implementation of Handler.handle

Defined in packages/core/src/http/http.ts:240

---

core.Http.constructor

constructor

• `Private` **new Http**()

Private **constructor** (which should never be called).

`Throws`

A TypeError upon construction.

Defined in packages/core/src/http/http.ts:227

---

core.Http

Http

• **Http**: `Object`

The **Http** namespace contains types and interfaces used and intended to be used in conjunction with the abstract Http class.

`See`

Http

Defined in packages/core/src/http/http.ts:12

packages/core/src/http/http.ts:56

---

core.Http.Handler

Handler

• **Handler**: `Object`

The **Handler** interface enforces the handle method with ajax compliant typing on the implementing class or object. This contract is used by the Proxy to type-guard the next hops.

Defined in packages/core/src/http/http.ts:33

Http.Handler.handle

handle

▸ **handle**(`request`): `Observable<Response<any>>`

Generic **handle** method enforcing ajax compliant typing. The method signature corresponds to that of the ajax method itself.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	Requesting Request.

Returns `Observable<Response<any>>`

An Observable of the requested Response.

Defined in packages/core/src/http/http.ts:42

---

core.Http.Request

Request

T **Request**: `AjaxConfig`

The **Request** type alias references the AjaxConfig interface and describes the shape of any Http **Request** parameters.

Defined in packages/core/src/http/http.ts:18

---

core.Http.Response

Response

T **Response**<T>: `AjaxResponse<T>`

The **Response** type alias references the AjaxResponse class and describes the shape of any Http **Response**.

Type parameters

Name	Type	Description
`T`	`any`	The **Response** type of a Request.

Defined in packages/core/src/http/http.ts:26

---

core.Kernel

Kernel

• **Kernel**: `Object`

Singleton **Kernel** class. The **Kernel** is essentially a dependency loader for ESM bundles (and their respective `importmaps`) or, depending on the runtime context and capabilities, UMD bundles and their transitive dependencies. By making use of the **Kernel**, applications based on the SGRUD client libraries may be comprised of multiple, optionally loaded Modules.

`Decorator`

Singleton

Defined in packages/core/src/kernel/kernel.ts:15

packages/core/src/kernel/kernel.ts:158

---

core.Kernel.[observable]

[observable][]

▸ **[observable]**(): `Subscribable<Module>`

Well-known `Symbol.observable` method returning a Subscribable. The returned Subscribable emits every Module that is successfully loaded.

Example

Subscribe to the loaded Modules:

```
import { Kernel } from '@sgrud/core';
import { from } from 'rxjs';

from(new Kernel()).subscribe(console.log);
```

Returns `Subscribable<Module>`

A Subscribable emitting loaded Modules.

Defined in packages/core/src/kernel/kernel.ts:264

---

core.Kernel.constructor

constructor

• **new Kernel**(`nodeModules?`)

Singleton **constructor**. The first time, this **constructor** is called, it will persist the nodeModules path Modules should be loaded from. Subsequent **constructor** calls will ignore this argument and return the Singleton instance. Through subscribing to the Subscribable returned by the well-known `Symbol.observable` method, the Module loading progress can be tracked.

Example

Instantiate the **Kernel** and require Modules:

```
import { Kernel } from '@sgrud/core';
import { forkJoin } from 'rxjs';

const kernel = new Kernel('https://unpkg.com');

forkJoin([
  kernel.require('example-module'),
  kernel.require('/static/local-module')
]).subscribe(console.log);
```

Parameters

Name	Type	Default value	Description
`nodeModules`	`string`	`'/node_modules'`	Optional location to load node modules from.

Defined in packages/core/src/kernel/kernel.ts:220

---

core.Kernel.insmod

insmod

▸ **insmod**(`module`, `source?`, `execute?`): `Observable<Module>`

Calling this method while supplying a valid `module` definition will chain the **ins**ert **mod**ule operations of the `module` dependencies and the `module` itself into an Observable, which is then returned. When multiple Modules are inserted, their dependencies are deduplicated by internally tracking all Modules and their transitive dependencies as separate loaders. Depending on the browser context, either the UMD

or ESM bundles (and their respective `importmaps`) are loaded via calling the script method. When **insmod**ding Modules which contain transitive sgrudDependencies, their compatibility is checked. Should a dependency version mismatch, the Observable returned by this method will throw.

`Throws`

An Observable RangeError or ReferenceError.

`Example`

insmod a Module by definition:

```
import { Kernel } from '@sgrud/core';
import packageJson from './module/package.json';

new Kernel().insmod(packageJson).subscribe(console.log);
```

Parameters

Name	Type	Default value	Description
`module`	`Module`	`undefined`	The Module definition to **insmod**.
`source`	`string`	`undefined`	An optional Module `source`.
`execute`	`boolean`	`false`	Wether to `execute` the Module.

Returns `Observable<Module>`

An Observable of the Module definition.

Defined in packages/core/src/kernel/kernel.ts:297

---

core.Kernel.nodeModules

nodeModules

• `Readonly` **nodeModules**: `string` = `'/node_modules'`

Optional location to load node modules from.

`Default Value`

`'/node_modules'`

Defined in packages/core/src/kernel/kernel.ts:227

---

core.Kernel.require

require

▸ **require**(`id`, `execute?`): `Observable<Module>`

requires a Module by name or source. If the supplied `id` is a relative path starting with `./`, an absolute path starting with `/` or an URL starting with `http`, the `id` is used as-is, otherwise it is appended to the nodeModules path and the `package.json` file within this path is retrieved via Http GET. The Module definition is then passed to the insmod method and returned.

`Example`

require a Module by `id`:

```
import { Kernel } from '@sgrud/core';

new Kernel().require('/static/lazy-module').subscribe(console.log);
```

Parameters

Name	Type	Default value	Description
`id`	`string`	`undefined`	The Module name or source to **require**.
`execute`	`boolean`	`true`	Wether to execute the Module.

Returns `Observable<Module>`

An Observable of the Module definition.

Defined in packages/core/src/kernel/kernel.ts:426

---

core.Kernel.resolve

resolve

▸ **resolve**(`name`, `source?`): `Observable<Module>`

resolves a Module definition by its `name`. The Module `name` is appended to the `source` path or, of none is supplied, the nodeModules path and the `package.json` file therein retrieved via Http GET. The parsed `package.json` is then emitted by the returned Observable.

`Example`

resolve a Module definition:

```
import { Kernel } from '@sgrud/core';

new Kernel().resolve('module').subscribe(console.log);
```

Parameters

Name	Type	Description
`name`	`string`	The Module `name` to **resolve**.
`source`	`string`	An optional Module `source`.

Returns `Observable<Module>`

An Observable of the Module definition.

Defined in packages/core/src/kernel/kernel.ts:460

---

core.Kernel.script

script

▸ **script**(`props`): `Observable<void>`

Inserts an HTMLScriptElement and applies the supplied `props` to it. The returned Observable emits and completes when the `onload` handler of the HTMLScriptElement is called. If no external `src` is supplied through the `props`, the `onload` handler of the element is called asynchronously. When the returned Observable completes, the inserted HTMLScriptElement is removed.

`Example`

Insert an HTMLScriptElement:

```
import { Kernel } from '@sgrud/core';

new Kernel().script({
  src: '/node_modules/module/bundle.js',
  type: 'text/javascript'
}).subscribe();
```

Parameters

Name	Type	Description
`props`	`Partial<HTMLScriptElement>`	Any properties to apply to the HTMLScriptElement.

Returns `Observable<void>`

An Observable of the HTMLScriptElements `onload`.

Defined in packages/core/src/kernel/kernel.ts:497

---

core.Kernel.verify

verify

▸ **verify**(`props`): `Observable<void>`

Inserts an HTMLLinkElement and applies the supplied `props` to it. This method is used to **verify** a Module bundle before importing and executing it by **verify**ing its Digest.

`Example`

verify a Module by Digest:

```
import { Kernel } from '@sgrud/core';

new Kernel().verify({
  href: '/node_modules/module/index.js',
  integrity: 'sha256-[...]',
  rel: 'modulepreload'
}).subscribe();
```

Parameters

Name	Type	Description
`props`	`Partial<HTMLLinkElement>`	Any properties to apply to the HTMLLinkElement.

Returns `Observable<void>`

An Observable of the appendage and removal of the element.

Defined in packages/core/src/kernel/kernel.ts:538

---

core.Kernel.changes

changes

• `Private` `Readonly` **changes**: `ReplaySubject<Module>`

Internal ReplaySubject tracking the loading state and therefore **changes** of loaded Modules. An Observable form of this internal ReplaySubject may be retrieved by invoking the well-known `Symbol.observable` method and subscribing to the returned Subscribable. The internal **changes** ReplaySubject emits all Module definitions loaded throughout the lifespan of this class.

Defined in packages/core/src/kernel/kernel.ts:169

---

core.Kernel.imports

imports

• `Private` `Readonly` **imports**: `Map<string, string>`

Internal Mapping to keep track of all via `importmaps` declared Module identifiers to their corresponding paths. This map is used for housekeeping, e.g., to prevent the same Module identifier to be defined multiple times.

Defined in packages/core/src/kernel/kernel.ts:177

---

core.Kernel.loaders

loaders

• `Private` `Readonly` **loaders**: `Map<string, ReplaySubject<Module>>`

Internal Mapping of all Modules **loaders** to a ReplaySubject. This ReplaySubject tracks the loading process as such, that it emits the Module definition once the respective Module is fully loaded (including dependencies etc.) and then completes.

Defined in packages/core/src/kernel/kernel.ts:186

---

core.Kernel.shimmed

shimmed

• `Private` `Readonly` **shimmed**: `string`

Internally used string to suffix the `importmap` and `module` types of HTMLScriptElements with, if applicable. This string is set to whatever trails the type of HTMLScriptElements encountered upon initialization, iff their type starts with `importmap`.

Defined in packages/core/src/kernel/kernel.ts:194

---

core.Kernel

Kernel

• **Kernel**: `Object`

The **Kernel** namespace contains types and interfaces used and intended to be used in conjunction with the Singleton Kernel class.

`See`

Kernel

Defined in packages/core/src/kernel/kernel.ts:15

packages/core/src/kernel/kernel.ts:158

---

core.Kernel.Digest

Digest

T **Digest**: ‘sha${256 | 384 | 512}-${string}‘

String literal helper type. Enforces any assigned string to represent a browser-parsable **Digest** hash. A **Digest** hash is used to represent a hash for Subresource Integrity validation.

`Example`

A valid **Digest**:

```ts
import { type Kernel } from '@sgrud/core';

const digest: Kernel.Digest = 'sha256-[...]';
```

Defined in packages/core/src/kernel/kernel.ts:30

---

core.Kernel.Module

Module

• **Module**: `Object`

Interface describing the shape of a **Module** while being aligned with well-known `package.json` fields. This interface additionally specifies optional `sgrudDependencies` and `webDependencies` mappings, which both are used by the Kernel to determine SGRUD module dependencies and runtime dependencies.

`Example`

An exemplary **Module** definition:

```ts
import { type Kernel } from '@sgrud/core';

const module: Kernel.Module = {
  name: 'module',
  version: '0.0.0',
  exports: './module.exports.js',
  unpkg: './module.unpkg.js',
  sgrudDependencies: {
    sgrudDependency: '^0.0.1'
  },
  webDependencies: {
    webDependency: {
      exports: {
        webDependency: './webDependency.exports.js'
      },
      unpkg: [
        './webDependency.unpkg.js'
      ]
    }
  }
};
```

Defined in packages/core/src/kernel/kernel.ts:65

Kernel.Module.digest

digest

• `Optional` `Readonly` **digest**: `Record<string, 'sha256-${string}' | 'sha384-${string}' | 'sha512-${string}'>`

Optional bundle Digests. If hashes are supplied, they will be used to verify the Subresource Integrity of the respective bundles.

Defined in packages/core/src/kernel/kernel.ts:93

Kernel.Module.exports

exports

• `Optional` `Readonly` **exports**: `string`

Optional ESM entry point.

Defined in packages/core/src/kernel/kernel.ts:81

Kernel.Module.name

name

• `Readonly` **name**: `string`

The **name** of the Module.

Defined in packages/core/src/kernel/kernel.ts:70

Kernel.Module.sgrudDependencies

sgrudDependencies

• `Optional` `Readonly` **sgrudDependencies**: `Record<string, string>`

Optional SGRUD Module dependencies.

Defined in packages/core/src/kernel/kernel.ts:98

Kernel.Module.unpkg

unpkg

• `Optional` `Readonly` **unpkg**: `string`

Optional UMD entry point.

Defined in packages/core/src/kernel/kernel.ts:86

Kernel.Module.version

version

• `Readonly` **version**: `string`

The Module version, formatted as according to the semver specifications.

Defined in packages/core/src/kernel/kernel.ts:76

Kernel.Module.webDependencies

webDependencies

• `Optional` `Readonly` **webDependencies**: `Record<string, WebDependency>`

Optional WebDependency mapping.

Defined in packages/core/src/kernel/kernel.ts:103

---

core.Kernel.WebDependency

WebDependency

• **WebDependency**: `Object`

Interface describing runtime dependencies of a Module. A Module may specify an array of UMD bundles to be loaded by the Kernel through the `unpkg` property. A Module may also specify a mapping of `import` specifiers to Module-relative paths through the `exports` property. Every specified **WebDependency** is loaded before respective bundles of the Module, which depends on the specified **WebDependency**, will be loaded themselves.

`Example`

An exemplary **webDependency** definition:

```
import { type Kernel } from '@sgrud/core';

const webDependency: Kernel.WebDependency = {
  exports: {
    webDependency: './webDependency.exports.js'
  },
  unpkg: [
    './webDependency.unpkg.js'
  ]
};
```

Defined in packages/core/src/kernel/kernel.ts:131

Kernel.WebDependency.exports

exports

• `Optional` `Readonly` **exports**: `Record<string, string>`

Optional ESM runtime dependencies.

Defined in packages/core/src/kernel/kernel.ts:136

Kernel.WebDependency.unpkg

unpkg

• `Optional` `Readonly` **unpkg**: `string[]`

Optional UMD runtime dependencies.

Defined in packages/core/src/kernel/kernel.ts:141

---

core.Linker

Linker

• **Linker**<K, V>: `Object`

The Singleton **Linker** class provides the means to lookup and retrieve instances of Targeted constructors. The **Linker** is used throughout the SGRUD client libraries, e.g., by the Factor decorator, to provide and retrieve different centrally provisioned class instances.

`Decorator`

Singleton

`Example`

Preemptively link an instance:

```ts
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>([
  [Service, new Service('linked')]
]);
```

Type parameters

Name	Type	Description
K	extends () => V	The Targeted constructor type.
V	InstanceType<K>	The Targeted InstanceType.

Defined in packages/core/src/linker/linker.ts:35

---

core.Linker.constructor

constructor

• **new Linker**<K, V>(entries?)

Type parameters

Name	Type
K	extends () => V
V	InstanceType<K>

Parameters

Name	Type
entries?	null | readonly readonly [K, V][]

Inherited from Map<K, V>.constructor

Defined in node_modules/typescript/lib/lib.es2015.collection.d.ts:51

• **new Linker**<K, V>(iterable?)

Type parameters

Name	Type
K	extends () => V
V	InstanceType<K>

Parameters

Name	Type
iterable?	null | Iterable<readonly [K, V]>

Inherited from Map<K, V>.constructor

Defined in node_modules/typescript/lib/lib.es2015.iterable.d.ts:159

---

core.Linker.get

get

▸ **get**(target): V

Overridden **get** method. Calling this method looks up the linked instance based on the supplied target constructor. If no linked instance is found, one is created by calling the new operator on the target constructor. Therefor the target constructors must not require parameters.

Example

Retrieve a linked instance:

```
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>().get(Service);
```

Parameters

Name	Type	Description
target	K	The target constructor for which to retrieve an instance.

Returns V

The already linked or a newly constructed and linked instance.

Overrides Map.get

Defined in packages/core/src/linker/linker.ts:58

---

core.Linker.getAll

getAll

▸ **getAll**(target): V[]

The **getAll** method returns all linked instances, which satisfy instanceof target. Use this method when multiple linked target constructors extend the same base class and are to be retrieved.

Example

Retrieve all linked instances of a Service:

```
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>().getAll(Service);
```

Parameters

Name	Type	Description
target	K	The target constructor for which to retrieve instances.

Returns V[]

All already linked instance of the target constructor.

Defined in packages/core/src/linker/linker.ts:85

---

core.Merge

Merge

T **Merge**<T>: `T` extends `T` ? (_: `T`) => `T` : `never` extends (_: infer I) => `T` ? `I` : `never`

Type helper to convert union types (`A | B`) to intersection types (`A & B`).

`Remarks`

https://github.com/microsoft/TypeScript/issues/29594

Type parameters

Name	Description
`T`	The union type to **Merge**.

Defined in

packages/core/src/typing/merge.ts:8

---

core.Mutable

Mutable

T **Mutable**<T>: { -readonly [K in keyof T]: T[K] }

Type helper marking the supplied type as **Mutable** (opposed to `readonly`).

`Remarks`

https://github.com/Microsoft/TypeScript/issues/24509

Type parameters

Name	Type	Description
`T`	extends `object`	The readonly type to make **Mutable**.

Defined in

packages/core/src/typing/mutable.ts:8

---

core.Provide

Provide

T **Provide**<K, V>: (...`args`: `any`[]) => `InstanceType`<V> & { [`provide`]: K extends `Registration` ? K : `Registration` }

Type helper enforcing the provide symbol property to contain a magic string (typed as Registration) on base constructors decorated with the corresponding Provide decorator. The **Provide** type helper is also used by the Provider decorator.

`See`

Provide

Type parameters

Name	Type	Description
`K`	extends `Registration`	The magic string Registration type.
`V`	extends (...`args`: `any`[]) => `InstanceType`<V>	The registered class constructor type.

Defined in packages/core/src/super/provide.ts:61

packages/core/src/super/provide.ts:19

---

core.Provide

Provide

▸ **Provide**<V, K>(): (constructor: V) => void

Class decorator factory. **Provide**s the decorated class to extending classes. Applying the **Provide** decorator enforces the Provide type which entails the declaration of a static provide property typed as Registration. The magic string assigned to this static property is used by the Provider factory function to get base classes from the Registry.

Example

Provide a base class:

```
import { Provide, provide } from '@sgrud/core';

@Provide()
export abstract class Base {

  public static readonly [provide] = 'sgrud.example.Base' as const;

}
```

See

- Provider
- Registry

Type parameters

Name	Type	Description
V	extends Provide<K, V>	The registered class constructor type.
K	extends '${string}.${string}.${string}' = V[typeof provide]	The magic string Registration type.

Returns fn

A class constructor decorator.

▸ (constructor): void

Parameters

Name	Type
constructor	V

Returns void

Defined in packages/core/src/super/provide.ts:61

---

core.Provider

Provider

▸ **Provider**<`V`, `K`>(`provider`): `V`

Provider of base classes. Extending this mixin-style function while supplying the `typeof` a Provided constructor enforces type safety and hinting on the supplied magic string and the resulting class which `extends` this **Provider** mixin. The main purpose of this pattern is bridging module gaps by de-coupling bundle files while maintaining a well-defined prototype chain. This still requires the base class to be defined (and Provided) before extension but allows intellisense'd OOP patterns across multiple modules while maintaining runtime language specifications.

`Example`

Extend a provided class:

```ts
import { Provider } from '@sgrud/core';
import { type Base } from 'example-module';

export class Class
  extends Provider<typeof Base>('sgrud.example.Base') {

  public constructor(...args: any[]) {
    super(...args);
  }

}
```

`See`

- Provide
- Registry

Type parameters

Name	Type	Description
`V`	extends `Provide`<`K`, `V`>	The registered class constructor type.
`K`	extends '${string}.${string}.${string}' = `V`[typeof `provide`]	The magic string Registration type.

Parameters

Name	Type	Description
`provider`	`K`	A magic string to retrieve the `provider` by.

Returns `V`

The constructor which Provides the Registration.

Defined in packages/core/src/super/provider.ts:64

---

core.Provider

Provider

• **Provider**<`V`>: `Object`

Type helper to allow referencing Provided constructors as `new`-able targets. Used and intended to be used in conjunction with the Provider decorator.

`See`

Provider

Type parameters

Name	Description
`V`	Instance type of the registered class constructor.

Defined in packages/core/src/super/provider.ts:64

packages/core/src/super/provider.ts:13

---

core.Provider.[provide]

[provide][]

• `Readonly` **[provide]**: ‘${string}.${string}.${string}‘

Enforced provider contract.

Defined in packages/core/src/super/provider.ts:18

---

core.Provider.constructor

constructor

• **constructor**: `Object`

---

core.Provider.constructor

constructor

• **new Provider**(`...args`)

Enforced constructor contract.

Parameters

Name	Type	Description
`...args`	`any[]`	The default class constructor rest parameter.

Defined in packages/core/src/super/provider.ts:25

---

core.Proxy

Proxy

• `Abstract` **Proxy**: `Object`

Abstract **Proxy** base class to implement Request interceptors, on the client side. By extending this abstract base class and providing the extending class to the Linker, e.g., by Targeting it, the class’s handle method will be called with the Request details (which could have been modified by a previous **Proxy**) and the next Handler, whenever a request is fired through the Http class.

`Decorator`

Provide

`Example`

Simple **Proxy** intercepting `file:` requests:

```ts
import { type Http, Provider, type Proxy, Target } from '@sgrud/core';
import { type Observable, of } from 'rxjs';
import { file } from './file';

@Target()
export class FileProxy
  extends Provider<typeof Proxy>('sgrud.core.Proxy') {

  public override handle(
    request: Http.Request,
    handler: Http.Handler
  ): Observable<Http.Response> {
    if (request.url.startsWith('file:')) {
      return of<Http.Response>(file);
    }

    return handler.handle(request);
  }

}
```

See

Http

Defined in packages/core/src/http/proxy.ts:44

---

core.Proxy.[provide]

[provide][]

■ `Static` `Readonly` **[provide]**: `"sgrud.core.Proxy"`

Magic string by which this class is provided.

See

provide

Defined in packages/core/src/http/proxy.ts:51

---

core.Proxy.constructor

constructor

• **new Proxy**()

---

core.Proxy.handle

handle

▸ `Abstract` **handle**(`request`, `handler`): `Observable<Response<any>>`

The **handle** method of linked classes extending the Proxy base class is called whenever an Request is fired. The extending class can either pass the `request` to the next `handler`, with or without modifying it, or an interceptor can chose to completely handle a `request` by itself through returning an Observable Response.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	The Request to be **handle**d.
`handler`	`Handler`	The next Handler to **handle** the `request`.

Returns `Observable<Response<any>>`

An Observable of the **handle**d Response.

Defined in packages/core/src/http/proxy.ts:64

---

core.Registration

Registration

T **Registration**: ‘${string}.${string}.${string}‘

String literal helper type. Enforces any assigned string to contain at least three dots. **Registration**s are used by the Registry to alias classes extending the base Provider as magic strings and should represent sane module paths in dot-notation.

`Example`

Library-wide **Registration** pattern:

```
import { type Registration } from '@sgrud/core';

const registration: Registration = 'sgrud.module.ClassName';
```

`See`

Registry

Defined in packages/core/src/super/registry.ts:19

---

core.Registry

Registry

• **Registry**<K, V>: `Object`

The Singleton **Registry** is a mapping used by the Provider to lookup Provided constructors by Registrations upon class extension. Magic strings should represent sane module paths in dot-notation. Whenever a currently not registered constructor is requested, an intermediary class is created, cached internally and returned. When the actual constructor is registered later, the previously created intermediary class is removed from the internal caching and further steps are taken to guarantee the transparent addressing of the actual constructor through the dropped intermediary class.

`Decorator`

Singleton

`See`

- Provide
- Provider

Type parameters

Name	Type	Description
K	extends Registration	The magic string Registration type.
V	extends (...args: any[]) => InstanceType<V>	The registered class constructor type.

Defined in packages/core/src/super/registry.ts:48

---

core.Registry.constructor

constructor

• **new Registry**<K, V>(tuples?)

Public **constructor**. The constructor of this class accepts the same parameters as its overridden `super` Map **constructor** and acts the same. I.e., through instantiating this Singleton class and passing a list of `tuples` of Registrations and their corresponding class constructors, these tuples may be preemptively registered.

Example

Preemptively provide a class constructor by magic string:

```
import { type Registration, Registry } from '@sgrud/core';
import { Service } from './service';

const registration = 'sgrud.example.Service';
new Registry<Registration, typeof Service>([
  [registration, Service]
]);
```

Type parameters

Name	Type
K	extends ‘${string}.${string}.${string}‘
V	extends (...args: any[]) => InstanceType<V>

Parameters

Name	Type	Description
tuples?	Iterable<[K, V]>	An Iterable of `tuples` provide.

Overrides Map<K, V>.constructor

Defined in packages/core/src/super/registry.ts:93

---

core.Registry.get

get

▸ **get**(registration): V

Overridden **get** method. Looks up the Provided constructor by magic string. If no provided constructor is found, an intermediary class is created, cached internally and returned. While this intermediary class and the functionality supporting it take care of inheritance, i.e., allow forward-referenced base classes to be extended, it cannot substitute for the actual extended constructor. Therefore, the static extension of forward-referenced classes is possible, but as long as the actual extended constructor is not registered (and therefore the intermediary class still caches the inheritance chain), the extending classes cannot be instantiated, called etc. Doing so will result in a ReferenceError being thrown.

Throws

A ReferenceError when a cached class is invoked.

Example

Retrieve a provided constructor by magic string:

```
import { type Registration, Registry } from '@sgrud/core';
import { type Service } from 'example-module';

const registration = 'sgrud.example.Service';
new Registry<Registration, typeof Service>().get(registration);
```

Parameters

Name	Type	Description
`registration`	`K`	The magic string to **get** the class constructor by.

Returns `V`

The Provided constructor or a cached intermediary.

Overrides Map.get

Defined in packages/core/src/super/registry.ts:133

---

core.Registry.set

set

▸ **set**(`registration`, `constructor`): `Registry<K, V>`

Overridden **set** method. Whenever a class constructor is provided by magic string through calling this method, a test is run, wether this constructor was previously requested and therefore cached as intermediary class. If so, the intermediary class is removed from the internal mapping and further steps are taken to guarantee the transparent addressing of the newly provided constructor through the previously cached and now dropped intermediary class.

`Example`

Preemptively provide a constructor by magic string:

```
import { type Registration, Registry } from '@sgrud/core';
import { Service } from './service';

const registration = 'sgrud.example.Service';
new Registry<Registration, typeof Service>().set(registration, Service);
```

Parameters

Name	Type	Description
`registration`	`K`	The magic string to **set** the class constructor by.
`constructor`	`V`	The `constructor` to register for the `registration`.

Returns `Registry<K, V>`

This Registry instance.

Overrides Map.set

Defined in packages/core/src/super/registry.ts:185

---

core.Registry.cached

cached

• `Private Readonly` **cached**: `Map<K, V>`

Internal Mapping of all **cached**, i.e., forward-referenced, class constructors. Whenever a constructor, which is not currently registered, is requested as a Provider, an intermediary class is created and stored within this map until the actual constructor is registered. As soon as this happens, the intermediary class is removed from this map and further steps are taken to guarantee the transparent addressing of the actual constructor through the dropped intermediary class.

Defined in packages/core/src/super/registry.ts:62

---

core.Registry.caches

caches

• `Private Readonly` **caches**: `WeakSet<V>`

Internally used WeakSet containing all intermediary classes created upon requesting a currently not registered constructor as provider. This set is used internally to check if a intermediary class has already been replaced by the actual constructor.

Defined in packages/core/src/super/registry.ts:70

---

core.Singleton

Singleton

▸ **Singleton**<T>(`apply?`): (`constructor`: T) => T

Class decorator factory. Enforces a transparent **Singleton** pattern on the decorated class. When calling the `new` operator on a decorated class for the first time, an instance of the decorated class is created using the supplied arguments, if any. This instance will remain the **Singleton** instance of the decorated class indefinitely. When calling the `new` operator on a decorated class already instantiated, the **Singleton** pattern is enforced and the previously constructed instance is returned. Instead, if provided, the `apply` callback is fired with the **Singleton** instance and the `new` invocation parameters.

`Example`

Singleton class:

```
import { Singleton } from '@sgrud/core';

@Singleton()
export class Service {}

new Service() === new Service(); // true
```

Type parameters

Name	Type	Description
T	extends (...`args`: `any`[]) => `any`	The type of the decorated constructor.

Parameters

Name	Type	Description
`apply?`	(`self`: `InstanceType<T>`, `args`: `ConstructorParameters<T>`) => `InstanceType<T>`	The callback to `apply` on subsequent `new` invocations.

Returns `fn`

A class constructor decorator.

▸ (`constructor`): T

Parameters

Name	Type
`constructor`	T

Returns `T`

Defined in packages/core/src/utility/singleton.ts:27

---

core.Spawn

Spawn

▸ **Spawn**(`worker`, `source?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

This prototype property decorator factory **Spawn**s a Worker and wraps and assigns the resulting Remote to the decorated prototype property.

`Example`

Spawn a Worker:

```
import { Spawn, type Thread } from '@sgrud/core';
import { type ExampleWorker } from 'example-worker';

export class ExampleWorkerHandler {

  @Spawn('example-worker')
  public readonly worker!: Thread<ExampleWorker>;

}
```

`See`

Thread

Parameters

Name	Type	Description
`worker`	`string` | `Endpoint` | `NodeEndpoint`	The `worker` module name or Endpoint to **Spawn**.
`source?`	`string`	An optional Module `source`.

Returns `fn`

A prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/core/src/thread/spawn.ts:32

---

core.Symbol

Symbol

▸ **Symbol**(`description?`): `symbol`

Proxy around the built-in Symbol object, returning the requested symbol or the name of the requested symbol prefixed with `'@@'`.

Parameters

Name	Type
`description?`	`string` | `number`

Returns `symbol`

Defined in node_modules/typescript/lib/lib.es2015.symbol.d.ts:29

---

core.Target

Target

T **Target**<V>: (...args: any[]) => V

Type parameters

Name	Description
V	The **Target**ed InstanceType.

Type declaration • (...args)

Type helper to allow Factoring **Target**ed constructors with required arguments. Used and to be used in conjunction with the Target decorator.

Parameters

Name	Type
...args	any[]

Defined in packages/core/src/linker/target.ts:56

packages/core/src/linker/target.ts:10

---

core.Target

Target

▸ **Target**<K>(factoryArgs?, target?): (constructor: K) => void

Class decorator factory. Links the **Target**ed constructor to its corresponding instance by applying the supplied `factoryArgs`. Employ this helper to link **Target**ed constructors with required arguments. Supplying a `target` constructor overrides its linked instance, if any, with the constructed instance.

Example

Target a service:

```
import { Target } from '@sgrud/core';

@Target(['default'])
export class Service {

  public constructor(
    public readonly param: string
  ) {}

}
```

Example

Factor a **Target**ed service:

```ts
import { Factor, type Target } from '@sgrud/core';
import { Service } from './service';

export class ServiceHandler {

  @Factor<Target<Service>>(() => Service)
  public readonly service!: Service;

}
```

See

- Factor
- Linker

Type parameters

Name	Type	Description
`K`	extends (...`args`: `any`[]) => `any`	The **Target**ed constructor type.

Parameters

Name	Type	Description
`factoryArgs?`	`ConstructorParameters<K>`	The arguments for the **Target**ed constructor.
`target?`	`K`	An optional **Target** constructor to override.

Returns `fn`

A class constructor decorator.

▸ (`constructor`): `void`

Parameters

Name	Type
`constructor`	`K`

Returns `void`

Defined in packages/core/src/linker/target.ts:56

---

core.Thread

Thread

T **Thread**<T>: `Promise<Remote<T>>`

Type alias describing an exposed class in a remote context. Represented by wrapping a Remote in a Promise. Used and intended to be used in conjunction with the Thread decorator.

See

Thread

Type parameters

Name	Description
T	The Remote **Thread** type.

Defined in packages/core/src/thread/thread.ts:32

packages/core/src/thread/thread.ts:13

---

core.Thread

Thread

▸ **Thread**(): (`constructor`: () => `any`) => `void`

Class decorator factory. exposes an instance of the decorated class as Worker **Thread**.

`Example`

ExampleWorker **Thread**:

```
import { Thread } from '@sgrud/core';

@Thread()
export class ExampleWorker {}
```

`See`

Spawn

Returns `fn`

A class constructor decorator.

▸ (`constructor`): `void`

Parameters

Name	Type
`constructor`	() => `any`

Returns `void`

Defined in packages/core/src/thread/thread.ts:32

---

core.Transit

Transit

• **Transit**: `Object`

The Targeted Singleton **Transit** class is a built-in Proxy intercepting all connections opened by the Http class. This Proxy implements the `Symbol.observable` pattern, through which it emits an array of all currently open Requests every time a new Request is fired or a previously fired Request completes.

`Decorator`

Target

`Decorator`

Singleton

`See`

- Http
- Proxy

Defined in packages/core/src/http/transit.ts:26

---

core.Transit.[provide]

[provide][]

■ `Static` `Readonly` **[provide]**: `"sgrud.core.Proxy"`

Magic string by which this class is provided.

See

provide

Inherited from Proxy.[provide]

Defined in packages/core/src/http/proxy.ts:51

---

core.Transit.[observable]

[observable][]

▸ **[observable]**(): `Subscribable<Response<any>[]>`

Well-known `Symbol.observable` method returning a Subscribable. The returned Subscribable emits all active Requests in an array, whenever this list changes. Using the returned Subscribable, e.g., a load indicator can easily be implemented.

Example

Subscribe to the currently active Request:

```
import { Transit, Linker } from '@sgrud/core';
import { from } from 'rxjs';

const transit = new Linker<typeof Transit>().get(Transit);
from(transit).subscribe(console.log);
```

Returns `Subscribable<Response<any>[]>`

A Subscribable emitting all active Request.

Defined in packages/core/src/http/transit.ts:70

---

core.Transit.constructor

constructor

• **new Transit**()

Public **constructor**. Called by the Target decorator to link this Proxy so it may be used by the Http class.

Overrides Proxy.constructor

Defined in packages/core/src/http/transit.ts:45

---

core.Transit.handle

handle

▸ **handle**(`request`, `handler`): `Observable<Response<any>>`

Overridden **handle** method of the Proxy base class. Mutates the `request` to also emit progress events while it is running. These progress events will be consumed by the Transit interceptor and re-supplied via the Subscribable returned by the `Symbol.observable` method.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	The Request to be **handle**d.
`handler`	`Handler`	The next Handler to **handle** the `request`.

Returns `Observable<Response<any>>`

An Observable of the **handle**d Response.

Overrides Proxy.handle

Defined in packages/core/src/http/transit.ts:84

---

core.Transit.changes

changes

• `Private Readonly` **changes**: `Subject<Transit>`

The **changes** Subject emits every time a request is added to or deleted from the internal requests mapping.

Defined in packages/core/src/http/transit.ts:33

---

core.Transit.requests

requests

• `Private Readonly` **requests**: `Map<AjaxConfig, Response<any>>`

Internal Mapping of all running requests. Mutating this map should be accompanied by an emittance of the changes Subject.

Defined in packages/core/src/http/transit.ts:39

---

core.TypeOf

TypeOf

• `Abstract` **TypeOf**: `Object`

Strict type-assertion and runtime type-checking utility. When type-checking variables in the global scope, e.g., `window` or `process`, make use of the `globalThis` object.

`Example`

Type-check global context:

```
import { TypeOf } from '@sgrud/core';

TypeOf.process(globalThis.process); // true if running in node context
TypeOf.window(globalThis.window);   // true if running in browser context
```

Defined in packages/core/src/utility/type-of.ts:15

---

core.TypeOf.array

array

▸ `Static` **array**(`value`): value is unknown[]

Type-check `value` for `unknown[]`.

`Example`

Type-check `null` for `unknown[]`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.array(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is unknown[]

Whether `value` is of type `unknown[]`.

Defined in packages/core/src/utility/type-of.ts:31

---

core.TypeOf.boolean

boolean

▸ `Static` **boolean**(`value`): value is boolean

Type-check `value` for `boolean`.

`Example`

Type-check `null` for `boolean`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.boolean(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is boolean

Whether `value` is of type `boolean`.

Defined in packages/core/src/utility/type-of.ts:49

---

core.TypeOf.date

date

▸ `Static` **date**(`value`): value is Date

Type-check `value` for `Date`.

`Example`

Type-check `null` for `Date`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.date(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is Date

Whether `value` is of type `Date`.

Defined in packages/core/src/utility/type-of.ts:67

---

core.TypeOf.function

function

▸ `Static` **function**(`value`): value is Function

Type-check `value` for `Function`.

`Example`

Type-check `null` for `Function`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.function(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is Function

Whether `value` is of type `Function`.

Defined in packages/core/src/utility/type-of.ts:85

---

core.TypeOf.null

null

▸ `Static` **null**(`value`): value is null

Type-check `value` for `null`.

`Example`

Type-check `null` for `null`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.null(null); // true
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is null

Whether `value` is of type `null`.

Defined in packages/core/src/utility/type-of.ts:103

---

core.TypeOf.number

number

▸ `Static` **number**(`value`): value is number

Type-check `value` for `number`.

`Example`

Type-check `null` for `number`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.number(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is number

Whether `value` is of type `number`.

Defined in packages/core/src/utility/type-of.ts:121

---

core.TypeOf.object

object

▸ `Static` **object**(`value`): value is object

Type-check `value` for `object`.

`Example`

Type-check `null` for `object`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.object(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is object

Whether `value` is of type `object`.

Defined in packages/core/src/utility/type-of.ts:139

---

core.TypeOf.process

process

▸ `Static` **process**(`value`): value is Process

Type-check `value` for `NodeJS.Process`.

`Example`

Type-check `null` for `NodeJS.Process`:

```
import { TypeOf } from '@sgrud/core';

TypeOf.process(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is Process

Whether `value` is of type `NodeJS.Process`.

Defined in packages/core/src/utility/type-of.ts:157

---

core.TypeOf.promise

promise

▸ `Static` **promise**(`value`): value is Promise<unknown>

Type-check `value` for `Promise<unknown>`.

`Example`

Type-check `null` for `Promise<unknown>`:

```
import { TypeOf } from '@sgrud/core';

TypeOf.promise(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is Promise<unknown>

Whether `value` is of type `Promise<unknown>`.

Defined in packages/core/src/utility/type-of.ts:175

---

core.TypeOf.regex

regex

▸ `Static` **regex**(`value`): value is RegExp

Type-check `value` for `RegExp`.

`Example`

Type-check `null` for `RegExp`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.regex(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is RegExp

Whether `value` is of type `RegExp`.

Defined in packages/core/src/utility/type-of.ts:193

---

core.TypeOf.string

string

▸ `Static` **string**(`value`): value is string

Type-check `value` for `string`.

`Example`

Type-check `null` for `string`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.string(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is string

Whether `value` is of type `string`.

Defined in packages/core/src/utility/type-of.ts:211

---

core.TypeOf.undefined

undefined

▸ `Static` **undefined**(`value`): value is undefined

Type-check `value` for `undefined`.

`Example`

Type-check `null` for `undefined`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.undefined(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is undefined

Whether `value` is of type `undefined`.

Defined in packages/core/src/utility/type-of.ts:229

---

core.TypeOf.url

url

▸ `Static` **url**(`value`): value is URL

Type-check `value` for `URL`.

`Example`

Type-check `null` for `URL`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.url(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is URL

Whether `value` is of type `URL`.

Defined in packages/core/src/utility/type-of.ts:247

---

core.TypeOf.window

window

▸ `Static` **window**(`value`): value is Window

Type-check `value` for `Window`.

`Example`

Type-check `null` for `Window`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.window(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	The `value` to type-check.

Returns value is Window

Whether `value` is of type `Window`.

Defined in packages/core/src/utility/type-of.ts:265

---

core.TypeOf.test

test

▸ `Static` `Private` **test**(`type`, `value`): `boolean`

Type-check `value` for `type`.

Parameters

Name	Type	Description
`type`	`string`	The `type` to check for.
`value`	`unknown`	The `value` to type-check.

Returns `boolean`

Whether `value` is `type`.

Defined in packages/core/src/utility/type-of.ts:276

---

core.TypeOf.constructor

constructor

• `Private` **new TypeOf**()

Private **constructor** (which should never be called).

`Throws`

A TypeError upon construction.

Defined in packages/core/src/utility/type-of.ts:285

---

core.assign

assign

▸ **assign**<T, S>(target, ...sources): T & Merge<S[number]>

assigns (deep copies) the values of all of the enumerable own properties from one or more sources to a target. The last value within the last sources object takes precedence over any previously encountered values.

Example

assign nested properties:

```
import { assign } from '@sgrud/core';

assign(
  { one: { one: true }, two: false },
  { one: { key: null } },
  { two: true }
);

// { one: { one: true, key: null }, two: true }
```

Type parameters

Name	Type	Description
T	extends Record<PropertyKey, any>	The type of the target object.
S	extends Record<PropertyKey, any>[]	The types of the sources objects.

Parameters

Name	Type	Description
target	T	The target object to **assign** properties to.
...sources	[...S[]]	An array of sources from which to deep copy properties.

Returns T & Merge<S[number]>

The **assign**ed-to target object.

Defined in packages/core/src/utility/assign.ts:29

---

core.pluralize

pluralize

▸ **pluralize**(singular): string

pluralizes words of the English language.

Example

Pluralize 'money':

```
import { pluralize } from '@sgrud/core';

pluralize('money'); // 'money'
```

Example

Pluralize 'thesis':

```
import { pluralize } from '@sgrud/core';

pluralize('thesis'); // 'theses'
```

Parameters

Name	Type	Description
`singular`	`string`	An English word in `singular` form.

Returns `string`

The **pluralize**d form of `singular`.

Defined in packages/core/src/utility/pluralize.ts:23

---

core.provide

provide

• `Const` **provide**: typeof `provide`

Unique symbol used as property key by the Provide type constraint.

Defined in packages/core/src/super/provide.ts:6

---

core.semver

semver

▸ **semver**(`version`, `range`): `boolean`

Best-effort **semver** matcher. The supplied `version` will be tested against the supplied `range`.

Example

Test `'1.2.3'` against `'>2 <1 || ~1.2.*'`:

```
import { semver } from '@sgrud/core';

semver('1.2.3', '>2 <1 || ~1.2.*'); // true
```

Example

Test `'1.2.3'` against `'~1.1'`:

```
import { semver } from '@sgrud/core';

semver('1.2.3', '~1.1'); // false
```

Parameters

Name	Type	Description
`version`	`string`	The to-be tested semantic `version` string.
`range`	`string`	The `range` to test the `version` against.

Returns `boolean`

Wether `version` satisfies `range`.

Defined in packages/core/src/kernel/semver.ts:25

Module: data

data

• **data**: `Object`

`@sgrud/data` - The SGRUD Data Model.

The functions and classes found within the **`@sgrud/data`** module are intended to ease the type safe data handling, i.e., retrieval, mutation and storage, within applications built upon the SGRUD client libraries. By extending the Model class and applying adequate decorators to the contained properties, the resulting extension will, in its static context, provide all necessary means to interact directly with the underlying repository, while the instance context of any class extending the abstract Model base class will inherit methods to observe changes to its instance field values, selectively complement the instance with fields from the backing data storage via type safe graph representations and to delete the respective instance from the data storage.

Defined in packages/data/index.ts:1

---

data.Enum

Enum

• `Abstract` **Enum**: `Object`

Abstract **Enum** helper class. This class is used by the Model to detect **Enum**erations within a Graph, as **Enum**erations (in contrast to plain strings) must not be quoted. This class should never be instantiated manually, but instead is used internally by the enumerate function.

`See`

enumerate

Defined in packages/data/src/model/enum.ts:10

---

data.Enum.constructor

constructor

• `Private` **new Enum**()

Private **constructor** (which should never be called).

`Throws`

A TypeError upon construction.

Overrides String.constructor

Defined in packages/data/src/model/enum.ts:18

---

data.HasMany

HasMany

▸ **HasMany**<`T`>(`typeFactory`, `transient?`): <`M`>(`model`: `M`, `field`: `Field`<`M`>) => `void`

Model field decorator factory. Using this decorator, Models can be enriched with one-to-many associations to other Models. The value for the `typeFactory` argument has to be another Model. By applying this decorator, the decorated field will (depending on the `transient` argument value) be taken into account when serializing or treemapping the Model containing the decorated field.

`Example`

Model with a one-to-many association:

```ts
import { HasMany, Model } from '@sgrud/data';
import { OwnedModel } from './owned-model';

export class ExampleModel extends Model<ExampleModel> {

  @HasMany(() => OwnedModel)
  public field?: OwnedModel[];

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasOne
- Property

Type parameters

Name	Type	Description
`T`	extends `Type<Model<any>, T>`	The field value constructor type.

Parameters

Name	Type	Default value	Description
`typeFactory`	() => `T`	`undefined`	A forward reference to the field value constructor.
`transient`	`boolean`	`false`	Whether the decorated field is `transient`.

Returns `fn`

A Model field decorator.

▸ `<M>(model, field): void`

Type parameters

Name	Type
`M`	extends `Model<any, M>`

Parameters

Name	Type
`model`	`M`
`field`	`Field<M>`

Returns `void`

Defined in packages/data/src/relation/has-many.ts:46

---

data.HasOne

HasOne

▸ **HasOne**<T>(typeFactory, transient?): <M>(model: M, field: Field<M>) => void

Model field decorator factory. Using this decorator, Models can be enriched with one-to-one associations to other Models. The value for the `typeFactory` argument has to be another Model. By applying this decorator, the decorated field will (depending on the `transient` argument value) be taken into account when serializing or treemapping the Model containing the decorated field.

Example

Model with a one-to-one association:

```
import { HasOne, Model } from '@sgrud/data';
import { OwnedModel } from './owned-model';

export class ExampleModel extends Model<ExampleModel> {

  @HasOne(() => OwnedModel)
  public field?: OwnedModel;

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasMany
- Property

Type parameters

Name	Type	Description
T	extends Type<Model<any>, T>	The field value constructor type.

Parameters

Name	Type	Default value	Description
typeFactory	() => T	undefined	A forward reference to the field value constructor.
transient	boolean	false	Whether the decorated field is transient.

Returns fn

A Model field decorator.

▸ <M>(model, field): void

Type parameters

Name	Type
M	extends Model<any, M>

Parameters

Name	Type
model	M
field	Field<M>

Returns void

Defined in packages/data/src/relation/has-one.ts:46

---

data.HttpQuerier

HttpQuerier

• **HttpQuerier**: `Object`

The **HttpQuerier** class implements an Http based Querier, i.e., an extension of the abstract Querier base class, allowing for Model queries to be executed via HTTP. To use this class, provide it to the Linker by either extending it, and decorating the extending class with the Target decorator, or by preemptively supplying an instance of this class to the Linker.

`Example`

Provide the **HttpQuerier** to the Linker:

```
import { Linker } from '@sgrud/core';
import { HttpQuerier } from '@sgrud/data';

new Linker<typeof HttpQuerier>([
  [HttpQuerier, new HttpQuerier('https://api.example.com')]
]);
```

`See`

- Model
- Querier

Defined in packages/data/src/querier/http.ts:28

---

data.HttpQuerier.[provide]

[provide][]

■ `Static` `Readonly` **[provide]**: `"sgrud.data.Querier"`

Magic string by which this class is provided.

`See`

provide

Inherited from Querier.[provide]

Defined in packages/data/src/querier/querier.ts:96

---

data.HttpQuerier.commit

commit

▸ **commit**(`operation`, `variables?`): `Observable<unknown>`

Overridden **commit** method of the Querier base class. When this Querier is made available via the Linker, this overridden **commit** method is called when this Querier claims the highest priority to **commit** an Operation, depending on the Model from which the Operation originates.

`Throws`

An Observable of an AggregateError.

Parameters

Name	Type	Description
`operation`	‘subscription ${string}‘ | ‘mutation ${string}‘ | ‘query ${string}‘	The Operation to be **commit**ted.
`variables?`	`Variables`	Any Variables within the `operation`.

Returns `Observable<unknown>`

An Observable of the committed Operation.

Overrides Querier.commit

Defined in packages/data/src/querier/http.ts:82

---

data.HttpQuerier.constructor

constructor

• **new HttpQuerier**(`endpoint`, `prioritize?`)

Public **constructor** consuming the HTTP `endpoint` Model queries should be committed against, and an dynamic or static `prioritize` value. The `prioritize` value may either be a mapping of Models to corresponding priorities or a static priority for this Querier.

Parameters

Name	Type	Default value	Description
`endpoint`	`string`	`undefined`	The HTTP `endpoint` to commit queries against.
`prioritize`	`number` | `Map<Type<Model<any>>, number>`	`0`	The dynamic or static prioritization.

Overrides Querier.constructor

Defined in packages/data/src/querier/http.ts:50

---

data.HttpQuerier.priority

priority

▸ **priority**(`model`): `number`

Overridden **priority** method of the Querier base class. When an Operation is to be committed, this method is called with the respective `model` Type and returns the claimed **priority** to commit this Model.

Parameters

Name	Type	Description
`model`	`Type<Model<any>>`	The Model to be committed.

Returns `number`

The numeric **priority** of this Querier implementation.

Overrides Querier.priority

Defined in packages/data/src/querier/http.ts:108

---

data.HttpQuerier.types

types

• `Readonly` **types**: `Set<Type>`

A set containing the Types this Querier can handle. As HTTP connections are short-lived, the HttpQuerier may only handle one-off **types**, namely `'mutation'` and `'query'`.

Overrides Querier.types

Defined in packages/data/src/querier/http.ts:36

---

data.HttpQuerier.endpoint

endpoint

• `Private Readonly` **endpoint**: `string`

The HTTP `endpoint` to commit queries against.

Defined in packages/data/src/querier/http.ts:55

---

data.HttpQuerier.prioritize

prioritize

• `Private Readonly` **prioritize**: `number | Map<Type<Model<any>>, number> = 0`

The dynamic or static prioritization.

`Default Value`

`0`

`See`

priority

Defined in packages/data/src/querier/http.ts:64

---

data.Model

Model

• `Abstract` **Model**`<M>`: `Object`

Abstract base class to implement data **Model**s. By extending this abstract base class while providing the `Symbol.toStringTag` property containing the singular name of the resulting data **Model**, type safe data handling, i.e., retrieval, mutation and storage, can easily be achieved. Through the use of the static- and instance-scoped polymorphic `this`, all inherited operations warrant type safety and provide intellisense.

`Example`

Extend the **Model** base class:

```
import { Model, Property } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  @Property(() => String)
  public field: string?;
```

```
  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

Querier

Type parameters

Name	Type	Description
`M`	extends `Model` = `any`	The extending **Model** InstanceType.

Defined in

packages/data/src/model/model.ts:18

packages/data/src/model/model.ts:126

packages/data/src/model/model.ts:323

---

data.Model.commit

commit

▸ `Static` **commit**<`T`>(`this`, `operation`, `variables?`): `Observable<unknown>`

Static **commit** method. Calling this method on a class extending the abstract Model base class, while supplying an `operation` and all its embedded `variables`, will dispatch the Operation to the respective Model repository through the highest priority Querier or, if no Querier is compatible, an error is thrown. This method is the entry point for all Model-related data transferral and is internally called by all other distinct methods of the Model.

Throws

An Observable ReferenceError on incompatibility.

Example

commit a `query`-type operation:

```
import { ExampleModel } from './example-model';

ExampleModel.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
}`, {
  variable: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`operation`	‘subscription ${string}‘ | ‘mutation ${string}‘ | ‘query ${string}‘	The Operation to be **commit**ted.

Name	Type	Description
`variables?`	`Variables`	Any Variables within the `operation`.

Returns `Observable<unknown>`

An Observable of the **commit**med `operation`.

Defined in packages/data/src/model/model.ts:357

---

data.Model.deleteAll

deleteAll

▸ `Static` **deleteAll**<`T`>(`this`, `uuids`): `Observable<unknown>`

Static **deleteAll** method. Calling this method on a class extending the Model, while supplying an array of `uuids`, will dispatch the deletion of all Model instances identified by these UUIDs to the respective Model repository by internally calling commit with suitable arguments. Through this method, bulk-deletions from the respective Model repository can be achieved.

`Example`

Drop all model instances by UUIDs:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteAll([
  'b050d63f-cede-46dd-8634-a80d0563ead8',
  'a0164132-cd9b-4859-927e-ba68bc20c0ae',
  'b3fca31e-95cd-453a-93ae-969d3b120712'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`uuids`	`string[]`	An array of `uuids` of Models to be deleted.

Returns `Observable<unknown>`

An Observable of the deletion.

Defined in packages/data/src/model/model.ts:410

---

data.Model.deleteOne

deleteOne

▸ `Static` **deleteOne**<`T`>(`this`, `uuid`): `Observable<unknown>`

Static **deleteOne** method. Calling this method on a class extending the Model, while supplying an `uuid`, will dispatch the deletion of the Model instance identified by this UUID to the respective repository by internally calling the commit operation with suitable arguments. Through this method, the deletion of a single Model instance from the respective Model repository can be achieved.

`Example`

Drop one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteOne(
  '18f3aa99-afa5-40f4-90c2-71a2ecc25651'
).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`uuid`	`string`	The `uuid` of the Model instance to be deleted.

Returns `Observable<unknown>`

An Observable of the deletion.

Defined in packages/data/src/model/model.ts:444

---

data.Model.findAll

findAll

▸ `Static` **findAll**<`T`>(`this`, `filter`, `graph`): `Observable<Results<T>>`

Static **findAll** method. Calling this method on a class extending the abstract Model base class, while supplying a `filter` to match Model instances by and a `graph` containing the fields to be included in the result, will dispatch a lookup operation to the respective repository by internally calling the commit operation with suitable arguments. Through this method, the bulk-lookup of Model instances from the respective Model repository can be achieved.

`Example`

Lookup all UUIDs for model instances modified between two dates:

```
import { ExampleModel } from './example-model';

ExampleModel.findAll({
  expression: {
    conjunction: {
      operands: [
        {
          entity: {
            operator: 'GREATER_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-01-01')
          }
```

```
        },
        {
          entity: {
            operator: 'LESS_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-12-12')
          }
        }
      ],
      operator: 'AND'
    }
  }
}, [
  'uuid',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`filter`	`Filter<T>`	A Filter to find Model instances by.
`graph`	`Graph<T>`	A Graph of fields to be returned.

Returns `Observable<Results<T>>`

An Observable of the find operation.

Defined in packages/data/src/model/model.ts:503

---

data.Model.findOne

findOne

▸ `Static` **findOne**`<T>(this, shape, graph): Observable<T>`

Static **findOne** method. Calling this method on a class extending the abstract Model base class, while supplying the `shape` to match the Model instance by and a `graph` describing the fields to be included in the result, will dispatch the lookup operation to the respective repository by internally calling the commit operation with suitable arguments. Through this method, the retrieval of one specific Model instance from the respective Model repository can be achieved.

`Example`

Lookup one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.findOne({
  id: '2cfe7609-c4d9-4e4f-9a8b-ad72737db48a'
}, [
  'uuid',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`shape`	`Shape<T>`	The Shape of instance to find.
`graph`	`Graph<T>`	A Graph of fields to be returned.

Returns `Observable<T>`

An Observable of the find operation.

Defined in packages/data/src/model/model.ts:552

---

data.Model.saveAll

saveAll

▸ `Static` **saveAll**`<T>(this, models, graph): Observable<T[]>`

Static **saveAll** method. Calling this method on a class extending the abstract Model base class, while supplying a list of `models` which to save and a `graph` describing the fields to be returned in the result, will dispatch the save operation to the respective Model repository by internally calling the commit operation with suitable arguments. Through this method, bulk-persistance of Model instances from the respective Model repository can be achieved.

`Example`

Persist multiple Models:

```
import { ExampleModel } from './example-model';

ExampleModel.saveAll([
  new ExampleModel({ field: 'example_1' }),
  new ExampleModel({ field: 'example_2' }),
  new ExampleModel({ field: 'example_3' })
], [
  'uuid',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`models`	`T[]`	An array of Models to be saved.

Name	Type	Description
`graph`	`Graph<T>`	The Graph of fields to be returned.

Returns `Observable<T[]>`

An Observable of the save operation.

Defined in packages/data/src/model/model.ts:598

---

data.Model.saveOne

saveOne

▸ `Static` **saveOne**<T>(`this`, `model`, `graph`): `Observable<T>`

Static **saveOne** method. Calling this method on a class extending the abstract Model base class, while supplying a `model` which to save and a `graph` describing the fields to be returned in the result, will dispatch the save operation to the respective Model repository by internally calling the commit operation with suitable arguments. Through this method, persistance of one specific Model instance from the respective Model repository can be achieved.

`Example`

Persist a model:

```
import { ExampleModel } from './example-model';

ExampleModel.saveOne(new ExampleModel({ field: 'example' }), [
  'uuid',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`model`	`T`	The Model which is to be saved.
`graph`	`Graph<T>`	A Graph of fields to be returned.

Returns `Observable<T>`

An Observable of the save operation.

Defined in packages/data/src/model/model.ts:640

---

data.Model.serialize

serialize

▸ Static **serialize**<T>(this, model, shallow?): undefined | Shape<T>

Static **serialize** method. Calling this method on a class extending the Model, while supplying a `model` which to **serialize** and optionally enabling `shallow` serialization, will return the **serialize**d Shape of the Model, i.e., a plain JSON representation of all Model fields, or `undefined`, if the supplied `model` does not contain any fields or values. By serializing `shallowly`, only such properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the serialization of one specific Model instance from the respective Model repository can be achieved.

Example

serialize a model:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const shape = ExampleModel.serialize(model);
console.log(shape); // { field: 'example' }
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	The extending Model InstanceType.

Parameters

Name	Type	Default value	Description
this	Type<T>	undefined	The explicit static polymorphic this parameter.
model	T	undefined	The Model which is to be **serialize**d.
shallow	boolean	false	Whether to **serialize** the Model shallowly.

Returns undefined | Shape<T>

The Shape of the Model or undefined.

Defined in packages/data/src/model/model.ts:683

---

data.Model.treemap

treemap

▸ Static **treemap**<T>(this, model, shallow?): undefined | Graph<T>

Static **treemap** method. Calling this method on a class extending the abstract Model base class, while supplying a `model` which to **treemap** and optionally enabling `shallow` **treemap**ping, will return a Graph describing the fields which are declared and defined on the supplied `model`, or `undefined`, if the supplied `model` does not contain any fields or values. By **treemap**ping `shallowly`, only properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the Graph for one specific Model instance from the respective Model repository can be retrieved.

Example

treemap a Model:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const graph = ExampleModel.treemap(model);
console.log(graph); // ['field']
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Default value	Description
`this`	`Type<T>`	`undefined`	The explicit static polymorphic `this` parameter.
`model`	`T`	`undefined`	The Model which is to be **treemap**ped.
`shallow`	`boolean`	`false`	Whether to **treemap** the Model `shallow`ly.

Returns `undefined | Graph<T>`

The Graph of the Model or `undefined`.

Defined in packages/data/src/model/model.ts:752

---

data.Model.unravel

unravel

▸ `Static` **unravel**<T>(`this`, `graph`): `string`

Static **unravel** method. Calling this method on a class extending the abstract Model base class, while supplying a `graph` describing the fields which to **unravel**, will return the Graph as raw string. Through this method, the Graph for one specific Model instance from the respective Model repository can be **unravel**ed into a raw string. This **unravel**ed Graph can then be consumed by, e.g., the commit method.

`Example`

unravel a Graph:

```
import { ExampleModel } from './example-model';

const unraveled = ExampleModel.unravel([
  'uuid',
  'modified',
  'field'
]);

console.log(unraveled); // '{id modified field}'
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`graph`	`Graph<T>`	A Graph which is to be **unravel**ed.

Returns `string`

The **unravel**ed Graph as raw string.

Defined in packages/data/src/model/model.ts:817

---

data.Model.valuate

valuate

▸ `Static` **valuate**<`T`>(`this`, `model`, `field`): `unknown`

Static **valuate** method. Calling this method on a class extending the abstract Model base class, while supplying a `model` and a `field` which to **valuate**, will return the preprocessed value (e.g., primitive representation of JavaScript Dates) of the supplied `field` of the supplied `model`. Through this method, the preprocessed `field` value of one specific Model instance from the respective Model repository can be retrieved.

`Example`

valuate a `field`:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ created: new Date(0) });
const value = ExampleModel.valuate(model, 'created');
console.log(value); // '1970-01-01T00:00:00.000+00:00'
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`model`	`T`	The Model which is to be **valuate**d.
`field`	`Field<T>`	A Field to be **valuate**d.

Returns `unknown`

The **valuate**d `field` value.

Defined in packages/data/src/model/model.ts:887

---

data.Model.[hasMany]

[hasMany][]

• `Optional` `Readonly` **[hasMany]**: `Record<keyof M, () => unknown>`

hasMany symbol property used by the HasMany decorator.

Defined in packages/data/src/model/model.ts:942

---

data.Model.[hasOne]

[hasOne][]

• `Optional` `Readonly` **[hasOne]**: `Record<keyof M, () => unknown>`

hasOne symbol property used by the HasOne decorator.

Defined in packages/data/src/model/model.ts:937

---

data.Model.[observable]

[observable][]

▸ **[observable]**(): `Subscribable<M>`

Well-known `Symbol.observable` method returning a Subscribable. The returned Subscribable emits all changes this Model instance experiences.

`Example`

Subscribe to a Model instance:

```
import { from } from 'rxjs';
import { ExampleModel } from './example-model';

const model = new ExampleModel();
from(model).subscribe(console.log);
```

Returns `Subscribable<M>`

A Subscribable emitting all Model changes.

Defined in packages/data/src/model/model.ts:1045

---

data.Model.[property]

[property][]

• `Optional` `Readonly` **[property]**: `Record<keyof M, () => unknown>`

property symbol property used by the Property decorator.

Defined in packages/data/src/model/model.ts:947

---

data.Model.assign

assign

▸ **assign**<T>(`this`, `...parts`): `Observable<T>`

Instance-scoped **assign** method. Calling this method, while supplying a list of `parts`, will **assign** all supplied `parts` to the Model instance. The **assign**ment is implemented as deep merge **assign**ment. Using this method, an existing Model instance can easily be mutated while still emitting the mutated changes.

`Example`

assign `parts` to a Model instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel();
model.assign({ field: 'example' }).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`T`	The explicit polymorphic `this` parameter.
`...parts`	`Shape<T>[]`	An array of `parts` to **assign** to this Model.

Returns `Observable<T>`

An Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1070

---

data.Model.clear

clear

▸ **clear**<T>(this, keys?): `Observable<T>`

Instance-scoped **clear** method. Calling this method on an instance of a class extending the abstract Model base class, while optionally supplying a list of `keys` which are to be **clear**ed, will set the value of the properties described by either the supplied `keys` or, if no `keys` were supplied, all enumerable properties of the class extending the abstract Model base class to `undefined`, effectively **clear**ing them.

`Example`

clear a Model instance selectively:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
model.clear(['field']).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`T`	The explicit polymorphic `this` parameter.
`keys?`	`Field<T>[]`	An optional array of `keys` to **clear**.

Returns `Observable<T>`

An Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1103

---

data.Model.commit

commit

▸ **commit**<T>(this, operation, variables?, mapping?): Observable<T>

Instance-scoped **commit** method. Internally calls the commit method on the static this-context of an instance of a class extending the abstract Model base class and furthermore assigns the returned data to the Model instance the **commit** method was called upon. When supplying a mapping, the returned data will be mutated through the supplied mapping (otherwise this mapping defaults to identity).

Example

commit a query-type operation:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel();

model.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
}`, {
  variable: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	The extending Model InstanceType.

Parameters

Name	Type	Description
this	T	The explicit polymorphic this parameter.
operation	‘subscription ${string}‘ | ‘mutation ${string}‘ | ‘query ${string}‘	The Operation to be **commit**ted.
variables?	Variables	Any Variables within the operation.
mapping	(next: unknown) => Shape<T>	An optional mutation to apply to the returned data.

Returns Observable<T>

An Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1160

---

data.Model.constructor

constructor

• **new Model**<M>(...parts)

Public **constructor**. The **constructor** of all classes extending the abstract Model base class, unless explicitly overridden, behaves analogous to the instance-scoped assign method, as it takes all supplied parts and assigns them to the instantiated and returned Model. The **constructor** furthermore wires some internal functionality, e.g., creates a new changes BehaviorSubject which emits every mutation this Model instance experiences etc.

Type parameters

Name	Type
`M`	extends `Model<any, M> = any`

Parameters

Name	Type	Description
`...parts`	`Shape<M>[]`	An array of `parts` to assign.

Defined in

packages/data/src/model/model.ts:1022

---

data.Model.created

created

• `Optional` **created**: `Date`

Transient creation Date of this Model instance.

`Decorator`

Property

Defined in

packages/data/src/model/model.ts:963

---

data.Model.delete

delete

▸ **delete**<T>(this): `Observable<T>`

Instance-scoped **delete** method. Internally calls the static deleteOne method while supplying the UUID of this instance of a class extending the abstract Model base class. Calling this method furthermore clears the Model instance and finalizes its deletion by completing the internal changes BehaviorSubject of the Model instance the **delete** method was called upon.

`Example`

delete a Model instance by UUID:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({
  id: '3068b30e-82cd-44c5-8912-db13724816fd'
});

model.delete().subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`T`	The explicit polymorphic `this` parameter.

Returns `Observable<T>`

An Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1196

---

data.Model.find

find

▸ **find**<T>(this, graph, shape?): `Observable<T>`

Instance-scoped **find** method. Internally calls the findOne method on the static `this`-context of an instance of a class extending the abstract Model base class and then assigns the returned data to the Model instance the **find** method was called upon.

`Example`

find a Model instance by UUID:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({
  id: '3068b30e-82cd-44c5-8912-db13724816fd'
});

model.find([
  'uuid',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends `Model<any, T> = M`	The extending Model InstanceType.

Parameters

Name	Type	Description
this	T	The explicit polymorphic `this` parameter.
graph	`Graph<T>`	A Graph of fields to be returned.
shape	`Shape<T>`	The Shape of the Model to find.

Returns `Observable<T>`

An Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1231

---

data.Model.modified

modified

• `Optional` **modified**: `Date`

Transient modification Date of this Model instance.

`Decorator`

Property

Defined in packages/data/src/model/model.ts:971

---

data.Model.save

save

▸ **save**<T>(this, graph?): Observable<T>

Instance-scoped **save** method. Internally calls the saveOne method on the static this-context of an instance of a class extending the abstract Model base class and then assigns the returned data to the Model instance the **save** method was called upon.

Example

save a Model instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });

model.save([
  'uuid',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	The extending Model InstanceType.

Parameters

Name	Type	Description
this	T	The explicit polymorphic this parameter.
graph	Graph<T>	A Graph of fields to be returned.

Returns Observable<T>

An Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1266

---

data.Model.serialize

serialize

▸ **serialize**<T>(this, shallow?): undefined | Shape<T>

Instance-scoped **serialize**er. Internally calls the serialize method on the static this-context of an instance of a class extending the abstract Model base class.

Example

serialize a Model instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
console.log(model.serialize()); // { field: 'example' }
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	The extending Model InstanceType.

Parameters

Name	Type	Default value	Description
`this`	`T`	`undefined`	The explicit polymorphic `this` parameter.
`shallow`	`boolean`	`false`	Whether to **serialize** `shallowly`.

Returns `undefined | Shape<T>`

The Shape of this instance or `undefined`.

Defined in packages/data/src/model/model.ts:1294

---

data.Model.treemap

treemap

▸ **treemap**<T>(this, shallow?): `undefined | Graph<T>`

Instance-scoped **treemap** method. Internally calls the treemap method on the static `this`-context of an instance of a class extending the abstract Model base class.

`Example`

treemap a Model instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
console.log(model.treemap()); // ['field']
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	The extending Model InstanceType.

Parameters

Name	Type	Default value	Description
`this`	`T`	`undefined`	The explicit polymorphic `this` parameter.
`shallow`	`boolean`	`false`	Whether to **treemap** `shallowly`.

Returns `undefined | Graph<T>`

A Graph of this instance or undefined.

Defined in packages/data/src/model/model.ts:1320

---

data.Model.uuid

uuid

• `Optional` **uuid**: `string`

UUID of this Model instance.

`Decorator`

Property

Defined in packages/data/src/model/model.ts:955

---

data.Model.[toStringTag]

[toStringTag][]

• `Protected` `Readonly` `Abstract` **[toStringTag]**: `string`

Enforced well-known `Symbol.toStringTag` property containing the singular name of this Model. The value of this property represents the repository which all instances of this Model are considered to belong to. In detail, the different operations committed through this Model are derived from this singular name (and the corresponding pluralized form).

`Example`

Provide a valid symbol property:

```
import { Model } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

Defined in packages/data/src/model/model.ts:932

---

data.Model.changes

changes

• `Protected` `Readonly` **changes**: `BehaviorSubject<M>`

BehaviorSubject emitting every time this Model instance experiences **changes**.

Defined in packages/data/src/model/model.ts:977

---

data.Model.entity

entity

• `Protected` `get` **entity**(): `string`

Accessor to the singular name of this Model.

Returns `string`

The singular name of this Model.

Defined in packages/data/src/model/model.ts:989

---

data.Model.plural

plural

• `Protected` `get` **plural**(): `string`

Accessor to the **plural**ized name of this Model.

Returns `string`

The **plural**ized name of this Model.

Defined in packages/data/src/model/model.ts:998

---

data.Model.static

static

• `Protected` `Readonly` **static**: `Type<M>`

Type-asserted alias for the **static** Model context.

Defined in packages/data/src/model/model.ts:982

---

data.Model.type

type

• `Protected` `get` **type**(): `string`

Accessor to the raw name of this Model.

Returns `string`

The raw name of this Model.

Defined in packages/data/src/model/model.ts:1007

---

data.Model

Model

• **Model**: `Object`

The **Model** namespace contains types and interfaces used and intended to be used in conjunction with classes extending the abstract Model base class. All the types and interfaces within this namespace are only applicable to classes extending the abstract Model base class, as their generic type argument is always constrained to this abstract base class.

See

Model

Defined in packages/data/src/model/model.ts:18

packages/data/src/model/model.ts:126

packages/data/src/model/model.ts:323

---

data.Model.Field

Field

T **Field**<T>: `string` & `Exclude`<keyof `T`, `Exclude`<keyof `Model`, `"uuid"` | `"created"` | `"modified"`>>

Type alias for all **Field**s, i.e., own enumerable properties (excluding internally used ones), of classes extending the abstract Model base class.

Type parameters

Name	Type	Description
`T`	extends `Model`	The extending Model InstanceType.

Defined in packages/data/src/model/model.ts:28

---

data.Model.Filter

Filter

T **Filter**<T>: `Params`<T>

Filter type alias referencing the Params type.

See

Params

Type parameters

Name	Type	Description
`T`	extends `Model`	The extending Model InstanceType.

Defined in packages/data/src/model/model.ts:38

packages/data/src/model/model.ts:126

---

data.Model.Filter

Filter

• **Filter**: `Object`

The **Filter** namespace contains types and interfaces to be used when searching through the repositories of classes extending the abstract Model base class. All the interfaces within this namespace are only applicable to classes extending the abstract Model base class, as their generic type argument is always constrained to this abstract base class.

See

Model

Defined in packages/data/src/model/model.ts:38

packages/data/src/model/model.ts:126

Model.Filter.Conjunction

Conjunction

T **Conjunction**: `"AND"` | `"AND_NOT"` | `"OR"` | `"OR_NOT"`

Type alias for a string union type of all possible **Conjunction**s, namely: `'AND'`, `'AND_NOT'`, `'OR'` and `'OR_NOT'`.

Defined in packages/data/src/model/model.ts:132

Model.Filter.Expression

Expression

• **Expression**<`T`>: `Object`

Interface describing the shape of an **Expression** which may be employed through the Params as part of a findAll. **Expression**s can either be the plain shape of an entity or compositions of multiple conjunctions.

Type parameters

Name	Type	Description
`T`	extends `Model`	The extending Model InstanceType.

Defined in packages/data/src/model/model.ts:160

Filter.Expression.conjunction

conjunction

• `Optional` `Readonly` **conjunction**: `Object`

conjunction of multiple filter Expressions requested data Models are matched against. The **conjunction** sibling parameter entity has to be undefined when supplying this parameter. By supplying filter Expressions, conjunct by specific Conjunction operators, fine-grained filter operations can be compiled.

Type declaration

Name	Type	Description
`operands`	`Expression<T>[]`	List of Expressions which are logically combined through an operator. These Expressions may be nested and can be used to construct complex composite filter operations.
`operator?`	`Conjunction`	Conjunction **operator** used to logically combine all supplied operands.

Defined in packages/data/src/model/model.ts:170

Filter.Expression.entity

entity

• `Optional` `Readonly` **entity**: `Object`

Shape the requested data Models are matched against. Supplying this parameter requires the conjunction sibling parameter to be `undefined`. By specifying the **entity** shape to match data Models against, simple filter operations can be compiled.

Type declaration

Name	Type	Description
`operator?`	`Operator`	Operator to use for matching.
`path`	`Path<T, []>`	Property **path** from within the data Model which to match against. The value which will be matched against has to be supplied through the value property.
`value`	`unknown`	Property **value** to match data Models against. The property path to this value has to be supplied through the path property.

Defined in

packages/data/src/model/model.ts:193

Model.Filter.Operator

Operator

T **Operator**: `"EQUAL"` | `"GREATER_OR_EQUAL"` | `"GREATER_THAN"` | `"LESS_OR_EQUAL"` | `"LESS_THAN"` | `"LIKE"` | `"NOT_EQUAL"`

Type alias for a string union type of all possible **Operator**s, namely: `'EQUAL'`, `'NOT_EQUAL'`, `'LIKE'`, `'GREATER_THAN'`, `'GREATER_OR_EQUAL'`, `'LESS_THAN'` and `'LESS_OR_EQUAL'`.

Defined in

packages/data/src/model/model.ts:143

Model.Filter.Params

Params

• **Params**<T>: `Object`

Interface describing the **Params** for the findAll method. This is the most relevant interface within this namespace (and is therefore also referenced by the Filter type alias), as it describes the input **Params** of any selective data retrieval.

See

Model

Type parameters

Name	Type	Description
`T`	extends `Model`	The extending Model InstanceType.

Defined in

packages/data/src/model/model.ts:228

Filter.Params.dir

dir

• `Optional` `Readonly` **dir**: `"desc"` | `"asc"`

Desired sorting **dir**ection of the requested data Models. To specify which field the results should be sorted by, the sort property must be supplied.

Defined in packages/data/src/model/model.ts:235

Filter.Params.expression

expression

• `Optional` `Readonly` **expression**: `Expression<T>`

Expression to evaluate results against. This **expression** may be a simple matching or more complex, conjunct and nested **expression**s.

Defined in packages/data/src/model/model.ts:242

Filter.Params.page

page

• `Optional` `Readonly` **page**: `number`

page number, i.e., offset within the list of all results for a data Model request. This property should be used together with the page size property.

Defined in packages/data/src/model/model.ts:249

Filter.Params.search

search

• `Optional` `Readonly` **search**: `string`

Free-text **search** field. This field overrides all expressions, as such that if this field contains a value, all expressions are ignored and only this free-text **search** filter is applied.

Defined in packages/data/src/model/model.ts:256

Filter.Params.size

size

• `Optional` `Readonly` **size**: `number`

Page **size**, i.e., number of results which should be included within the response to a data Model request. This property should be used together with the page offset property.

Defined in packages/data/src/model/model.ts:263

Filter.Params.sort

sort

• `Optional` `Readonly` **sort**: `Path<T, []>`

Property path used to determine the value which to **sort** the requested data Models by. This property should be used together with the sorting direction property.

Defined in packages/data/src/model/model.ts:270

Model.Filter.Results

Results

• **Results**<T>: `Object`

Interface describing the shape of Filtered **Results**. When invoking the findAll method, an Observable of this interface shape is returned.

Type parameters

Name	Type
T	extends Model

Defined in packages/data/src/model/model.ts:279

Filter.Results.result

result

• **result**: T[]

An array of Models representing the Filtered **results**.

Defined in packages/data/src/model/model.ts:284

Filter.Results.total

total

• **total**: number

The **total** number of Results, useful for the implementation of a pageable representation of Filtered Results.

Defined in packages/data/src/model/model.ts:290

---

data.Model.Graph

Graph

T **Graph**<T>: { [K in Field<T>]?: Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> | Model<infer I>[] ? Record<K, Graph<I> | Function> : K }[Field<T>][]

Mapped type to compile strongly typed **Graph**s of classes extending the abstract Model base class, while providing intellisense.

Type parameters

Name	Type	Description
T	extends Model	The extending Model InstanceType.

Defined in packages/data/src/model/model.ts:46

---

data.Model.Path

Path

T **Path**<T, N>: { [K in Field<T>]: N extends Object ? never : Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> | Model<infer I>[] ? `${K}.${Path<I, [...N, string]>}` : K }[Field<T>]

Mapped type to compile strongly typed property **Path**s of classes extending the abstract Model base class, while providing intellisense.

Type parameters

Name	Type	Description
T	extends Model	The extending Model InstanceType.
N	extends string[] = []	A string array type used to determine recursive depth.

Defined in

packages/data/src/model/model.ts:63

---

data.Model.Shape

Shape

T **Shape**<T>: { [K in Field<T>]?: Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> ? Shape<I> : Required<T>[K] extends Model<infer I>[] ? Shape<I>[] : Required<T>[K] }

Mapped type to compile strongly typed **Shape**s of classes extending the abstract Model base class, while providing intellisense.

Type parameters

Name	Type	Description
T	extends Model	The extending Model InstanceType.

Defined in

packages/data/src/model/model.ts:80

---

data.Model.Type

Type

• **Type**<T>: Object

Interface describing the **Type**, i.e., static constructable context, of classes extending the abstract Model base class.

Type parameters

Name	Type	Description
T	extends Model	The extending Model InstanceType.

Defined in

packages/data/src/model/model.ts:97

Model.Type.commit

commit

▸ **commit**<T>(this, operation, variables?): Observable<unknown>

Static **commit** method. Calling this method on a class extending the abstract Model base class, while supplying an operation and all its embedded variables, will dispatch the Operation to the respective Model repository through the highest priority Querier or, if no Querier is compatible, an error is thrown. This method is the entry point for all Model-related data transferral and is internally called by all other distinct methods of the Model.

Throws

An Observable ReferenceError on incompatibility.

Example

commit a query-type operation:

```
import { ExampleModel } from './example-model';

ExampleModel.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
}`, {
  variable: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	The extending Model InstanceType.

Parameters

Name	Type	Description
this	Type<T>	The explicit static polymorphic this parameter.
operation	‘subscription ${string}‘ | ‘mutation ${string}‘ | ‘query ${string}‘	The Operation to be **commit**ted.
variables?	Variables	Any Variables within the operation.

Returns Observable<unknown>

An Observable of the **commit**med operation.

Inherited from Required.commit

Defined in packages/data/src/model/model.ts:357

Model.Type.constructor

constructor

• **new Type**(...args)

Overridden and concretized constructor signature.

Parameters

Name	Type	Description
`...args`	`Shape<Model<any>>[]`	The default class constructor rest parameter.

Inherited from

Required<typeof Model>.constructor

Defined in

packages/data/src/model/model.ts:109

Model.Type.deleteAll

deleteAll

▸ **deleteAll**<`T`>(`this`, `uuids`): `Observable<unknown>`

Static **deleteAll** method. Calling this method on a class extending the Model, while supplying an array of `uuids`, will dispatch the deletion of all Model instances identified by these UUIDs to the respective Model repository by internally calling commit with suitable arguments. Through this method, bulk-deletions from the respective Model repository can be achieved.

`Example`

Drop all model instances by UUIDs:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteAll([
  'b050d63f-cede-46dd-8634-a80d0563ead8',
  'a0164132-cd9b-4859-927e-ba68bc20c0ae',
  'b3fca31e-95cd-453a-93ae-969d3b120712'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`uuids`	`string[]`	An array of `uuids` of Models to be deleted.

Returns

`Observable<unknown>`

An Observable of the deletion.

Inherited from

Required.deleteAll

Defined in

packages/data/src/model/model.ts:410

Model.Type.deleteOne

deleteOne

▸ **deleteOne**<T>(this, uuid): Observable<unknown>

Static **deleteOne** method. Calling this method on a class extending the Model, while supplying an uuid, will dispatch the deletion of the Model instance identified by this UUID to the respective repository by internally calling the commit operation with suitable arguments. Through this method, the deletion of a single Model instance from the respective Model repository can be achieved.

Example

Drop one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteOne(
  '18f3aa99-afa5-40f4-90c2-71a2ecc25651'
).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	The extending Model InstanceType.

Parameters

Name	Type	Description
this	Type<T>	The explicit static polymorphic this parameter.
uuid	string	The uuid of the Model instance to be deleted.

Returns Observable<unknown>

An Observable of the deletion.

Inherited from Required.deleteOne

Defined in packages/data/src/model/model.ts:444

Model.Type.findAll

findAll

▸ **findAll**<T>(this, filter, graph): Observable<Results<T>>

Static **findAll** method. Calling this method on a class extending the abstract Model base class, while supplying a filter to match Model instances by and a graph containing the fields to be included in the result, will dispatch a lookup operation to the respective repository by internally calling the commit operation with suitable arguments. Through this method, the bulk-lookup of Model instances from the respective Model repository can be achieved.

Example

Lookup all UUIDs for model instances modified between two dates:

```
import { ExampleModel } from './example-model';

ExampleModel.findAll({
  expression: {
    conjunction: {
      operands: [
        {
          entity: {
            operator: 'GREATER_OR_EQUAL',
```

```
            path: 'modified',
            value: new Date('2021-01-01')
          }
        },
        {
          entity: {
            operator: 'LESS_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-12-12')
          }
        }
      ],
      operator: 'AND'
    }
  }
}, [
  'uuid',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`filter`	`Filter<T>`	A Filter to find Model instances by.
`graph`	`Graph<T>`	A Graph of fields to be returned.

Returns `Observable<Results<T>>`

An Observable of the find operation.

Inherited from Required.findAll

Defined in packages/data/src/model/model.ts:503

Model.Type.findOne

findOne

▸ **findOne**<T>(this, shape, graph): `Observable<T>`

Static **findOne** method. Calling this method on a class extending the abstract Model base class, while supplying the `shape` to match the Model instance by and a `graph` describing the fields to be included in the result, will dispatch the lookup operation to the respective repository by internally calling the commit operation with suitable arguments. Through this method, the retrieval of one specific Model instance from the respective Model repository can be achieved.

`Example`

Lookup one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.findOne({
  id: '2cfe7609-c4d9-4e4f-9a8b-ad72737db48a'
}, [
  'uuid',
```

```
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`shape`	`Shape<T>`	The Shape of instance to find.
`graph`	`Graph<T>`	A Graph of fields to be returned.

Returns `Observable<T>`

An Observable of the find operation.

Inherited from Required.findOne

Defined in packages/data/src/model/model.ts:552

Model.Type.prototype

prototype

• `Readonly` **prototype**: `T`

Overridden `prototype` signature.

Overrides Required.prototype

Defined in packages/data/src/model/model.ts:102

Model.Type.saveAll

saveAll

▸ **saveAll**<`T`>(`this`, `models`, `graph`): `Observable<T[]>`

Static **saveAll** method. Calling this method on a class extending the abstract Model base class, while supplying a list of `models` which to save and a `graph` describing the fields to be returned in the result, will dispatch the save operation to the respective Model repository by internally calling the commit operation with suitable arguments. Through this method, bulk-persistance of Model instances from the respective Model repository can be achieved.

`Example`

Persist multiple Models:

```
import { ExampleModel } from './example-model';

ExampleModel.saveAll([
  new ExampleModel({ field: 'example_1' }),
  new ExampleModel({ field: 'example_2' }),
  new ExampleModel({ field: 'example_3' })
], [
```

```
  'uuid',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`models`	`T[]`	An array of Models to be saved.
`graph`	`Graph<T>`	The Graph of fields to be returned.

Returns `Observable<T[]>`

An Observable of the save operation.

Inherited from Required.saveAll

Defined in packages/data/src/model/model.ts:598

Model.Type.saveOne

saveOne

▸ **saveOne**<T>(`this`, `model`, `graph`): `Observable<T>`

Static **saveOne** method. Calling this method on a class extending the abstract Model base class, while supplying a `model` which to save and a `graph` describing the fields to be returned in the result, will dispatch the save operation to the respective Model repository by internally calling the commit operation with suitable arguments. Through this method, persistance of one specific Model instance from the respective Model repository can be achieved.

`Example`

Persist a model:

```
import { ExampleModel } from './example-model';

ExampleModel.saveOne(new ExampleModel({ field: 'example' }), [
  'uuid',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	The extending Model InstanceType.

Parameters

Name	Type	Description
this	Type<T>	The explicit static polymorphic this parameter.
model	T	The Model which is to be saved.
graph	Graph<T>	A Graph of fields to be returned.

Returns Observable<T>

An Observable of the save operation.

Inherited from Required.saveOne

Defined in packages/data/src/model/model.ts:640

Model.Type.serialize

serialize

▸ **serialize**<T>(this, model, shallow?): undefined | Shape<T>

Static **serialize** method. Calling this method on a class extending the Model, while supplying a model which to **serialize** and optionally enabling shallow serialization, will return the **serialize**d Shape of the Model, i.e., a plain JSON representation of all Model fields, or undefined, if the supplied model does not contain any fields or values. By serializing shallowly, only such properties defined on the supplied model are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the serialization of one specific Model instance from the respective Model repository can be achieved.

Example

serialize a model:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const shape = ExampleModel.serialize(model);
console.log(shape); // { field: 'example' }
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	The extending Model InstanceType.

Parameters

Name	Type	Default value	Description
this	Type<T>	undefined	The explicit static polymorphic this parameter.
model	T	undefined	The Model which is to be **serialize**d.
shallow	boolean	false	Whether to **serialize** the Model shallowly.

Returns undefined | Shape<T>

The Shape of the Model or undefined.

Inherited from Required.serialize

Defined in packages/data/src/model/model.ts:683

Model.Type.treemap

treemap

▸ **treemap**<T>(this, model, shallow?): undefined | Graph<T>

Static **treemap** method. Calling this method on a class extending the abstract Model base class, while supplying a `model` which to **treemap** and optionally enabling `shallow` **treemap**ping, will return a Graph describing the fields which are declared and defined on the supplied `model`, or `undefined`, if the supplied `model` does not contain any fields or values. By **treemap**ping `shallow`ly, only properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the Graph for one specific Model instance from the respective Model repository can be retrieved.

Example

treemap a Model:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const graph = ExampleModel.treemap(model);
console.log(graph); // ['field']
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	The extending Model InstanceType.

Parameters

Name	Type	Default value	Description
this	Type<T>	undefined	The explicit static polymorphic `this` parameter.
model	T	undefined	The Model which is to be **treemap**ped.
shallow	boolean	false	Whether to **treemap** the Model `shallow`ly.

Returns undefined | Graph<T>

The Graph of the Model or `undefined`.

Inherited from Required.treemap

Defined in packages/data/src/model/model.ts:752

Model.Type.unravel

unravel

▸ **unravel**<T>(this, graph): string

Static **unravel** method. Calling this method on a class extending the abstract Model base class, while supplying a `graph` describing the fields which to **unravel**, will return the Graph as raw string. Through this method, the Graph for one specific Model instance from the respective Model repository can be **unravel**ed into a raw string. This **unravel**ed Graph can then be consumed by, e.g., the commit method.

Example

unravel a Graph:

```
import { ExampleModel } from './example-model';

const unraveled = ExampleModel.unravel([
  'uuid',
  'modified',
  'field'
]);

console.log(unraveled); // '{id modified field}'
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	The extending Model InstanceType.

Parameters

Name	Type	Description
this	Type<T>	The explicit static polymorphic this parameter.
graph	Graph<T>	A Graph which is to be **unravel**ed.

Returns `string`

The **unravel**ed Graph as raw string.

Inherited from Required.unravel

Defined in packages/data/src/model/model.ts:817

Model.Type.valuate

valuate

▸ **valuate**<T>(this, model, field): unknown

Static **valuate** method. Calling this method on a class extending the abstract Model base class, while supplying a `model` and a `field` which to **valuate**, will return the preprocessed value (e.g., primitive representation of JavaScript Dates) of the supplied `field` of the supplied `model`. Through this method, the preprocessed `field` value of one specific Model instance from the respective Model repository can be retrieved.

Example

valuate a `field`:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ created: new Date(0) });
const value = ExampleModel.valuate(model, 'created');
console.log(value); // '1970-01-01T00:00:00.000+00:00'
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	The extending Model InstanceType.

Parameters

Name	Type	Description
`this`	`Type<T>`	The explicit static polymorphic `this` parameter.
`model`	`T`	The Model which is to be **valuate**d.
`field`	`Field<T>`	A Field to be **valuate**d.

Returns `unknown`

The **valuate**d `field` value.

Inherited from Required.valuate

Defined in packages/data/src/model/model.ts:887

---

data.Property

Property

T **Property**: typeof `Boolean` | typeof `Date` | typeof `Number` | typeof `Object` | typeof `String`

Type alias for a union type of all primitive constructors which may be used as `typeFactory` argument for the Property decorator.

See

Property

Defined in packages/data/src/relation/property.ts:61

packages/data/src/relation/property.ts:10

---

data.Property

Property

▸ **Property**<T>(`typeFactory`, `transient?`): <M>(`model`: `M`, `field`: `Field<M>`) => `void`

Model field decorator factory. Using this decorator, Models can be enriched with primitive fields. The compatible primitives are the subset of primitives JavaScript shares with JSON, i.e., Boolean, Date (serialized), Number and String. Objects cannot be uses as a `typeFactory` argument value, as Model fields containing objects should be declared by the HasOne and HasMany decorators. By employing this decorator, the decorated field will (depending on the `transient` argument value) be taken into account when serializing or treemapping the Model containing the decorated field.

Example

Model with a primitive field:

```
import { Model, Property } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  @Property(() => String)
  public field?: string;

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasOne
- HasMany

Type parameters

Name	Type	Description
`T`	extends `Property`	The field value constructor type.

Parameters

Name	Type	Default value	Description
`typeFactory`	() => `T`	`undefined`	A forward reference to the field value constructor.
`transient`	`boolean`	`false`	Whether the decorated field is `transient`.

Returns `fn`

A Model field decorator.

▸ `<M>(model, field): void`

Type parameters

Name	Type
`M`	extends `Model<any, M>`

Parameters

Name	Type
`model`	`M`
`field`	`Field<M>`

Returns `void`

Defined in packages/data/src/relation/property.ts:61

---

data.Querier

Querier

• `Abstract` **Querier**: `Object`

Abstract **Querier** base class to implement Model **Querier**s. By extending this abstract base class and providing the extending class to the Linker, e.g., by Targeting it, the priority method of the resulting class will be called whenever the Model requests or persists data and, if this class claims the highest priority, its commit method will be called.

`Decorator`

Provide

`Example`

Simple **Querier** stub:

```
import { Provider, Target } from '@sgrud/core';
import { type Querier } from '@sgrud/data';
import { type Observable } from 'rxjs';

@Target()
export class ExampleQuerier
```

```
  extends Provider<typeof Querier>('sgrud.data.Querier') {

  public override readonly types: Set<Querier.Type> = new Set<Querier.Type>([
    'query'
  ]);

  public override commit(
    operation: Querier.Operation,
    variables: Querier.Variables
  ): Observable<unknown> {
    throw new Error('Stub!');
  }

  public override priority(): number {
    return 0;
  }

}
```

See

Model

Defined in packages/data/src/querier/querier.ts:12

packages/data/src/querier/querier.ts:89

---

data.Querier.[provide]

[provide][]

■ `Static Readonly` **[provide]**: `"sgrud.data.Querier"`

Magic string by which this class is provided.

See

provide

Defined in packages/data/src/querier/querier.ts:96

---

data.Querier.commit

commit

▸ `Abstract` **commit**(`operation`, `variables?`): `Observable<unknown>`

The overridden **commit** method of Targeted Queriers is called by the Model to execute Operations. The invocation arguments are the `operation`, unraveled into a string, and all `variables` embedded within this operation. The extending class has to serialize the Variables and handle the operation. It's the callers responsibility to unravel the Operation prior to invoking this method, and to deserialize and (error) handle whatever response is received.

Parameters

Name	Type	Description
`operation`	'subscription ${string}' | 'mutation ${string}' | 'query ${string}'	The Operation to be **commit**ted.
`variables?`	`Variables`	Any Variables within the Operation.

Returns `Observable<unknown>`

An Observable of the **commit**ted Operation.

Defined in packages/data/src/querier/querier.ts:118

---

data.Querier.constructor

constructor

• **new Querier**()

---

data.Querier.priority

priority

▸ `Abstract` **priority**(`model`): `number`

When the Model executes Operations, all Targeted and compatible Queriers, i.e., implementations of the this class capable of handling the specific Type of the Operation to commit, will be asked to prioritize themselves regarding the respective Model. The querier claiming the highest **priority** will be chosen and its commit method called.

Parameters

Name	Type	Description
`model`	`Type<Model<any>>`	The Model to be committed.

Returns `number`

The numeric **priority** of this Querier implementation.

Defined in packages/data/src/querier/querier.ts:134

---

data.Querier.types

types

• `Readonly` `Abstract` **types**: `Set<Type>`

A set containing all **types** of queries this Querier can handle. May contain any of the `'mutation'`, `'query'` and `'subscription'` Types.

Defined in packages/data/src/querier/querier.ts:103

---

data.Querier

Querier

• **Querier**: `Object`

Querier namespace containing types and interfaces used and intended to be used in conjunction with the abstract Querier base class and in context of the Model data handling.

See

Querier

Defined in packages/data/src/querier/querier.ts:12

packages/data/src/querier/querier.ts:89

---

data.Querier.Operation

Operation

T **Operation**: ‘${Type} ${string}‘

String literal helper type. Enforces any assigned string to conform to the standard form of an **Operation**: A string starting with the Type, followed by one whitespace and the operation content.

Defined in packages/data/src/querier/querier.ts:28

---

data.Querier.Type

Type

T **Type**: `"mutation"` | `"query"` | `"subscription"`

Type alias for a string union type of all known Operation **Type**s: `'mutation'`, `'query'` and `'subscription'`.

Defined in packages/data/src/querier/querier.ts:18

---

data.Querier.Variables

Variables

• **Variables**: `Object`

Interface describing the shape of **Variables** which may be embedded within Operations. **Variables** are a simple key-value map, which can be deeply nested.

Defined in packages/data/src/querier/querier.ts:35

---

data.enumerate

enumerate

▸ **enumerate**<T>(`enumerator`): `T`

enumerate helper function. Enumerations are special objects and all used TypeScript `enum`s have to be looped through this helper function before they can be utilized in conjunction with the Model.

`Example`

enumerate a TypeScript enumeration:

```
import { enumerate } from '@sgrud/data';

enum Enumeration {
  One = 'ONE',
  Two = 'TWO'
}

export type ExampleEnum = Enumeration;
export const ExampleEnum = enumerate(Enumeration);
```

`See`

Model

Type parameters

Name	Type	Description
T	extends `object`	The type of TypeScript `enum`.

Parameters

Name	Type	Description
`enumerator`	`T`	The TypeScript `enum` to **enumerate**.

Returns `T`

The processed enumeration to be used by the Model.

Defined in packages/data/src/model/enum.ts:49

---

data.hasMany

hasMany

• `Const` **hasMany**: typeof `hasMany`

Unique symbol used as property key by the HasMany decorator to register decorated Model fields for further computation, e.g., serialization, treemapping etc.

See

HasMany

Defined in packages/data/src/relation/has-many.ts:11

---

data.hasOne

hasOne

• `Const` **hasOne**: typeof `hasOne`

Unique symbol used as property key by the HasOne decorator to register decorated Model fields for further computation, e.g., serialization, treemapping etc.

See

HasOne

Defined in packages/data/src/relation/has-one.ts:11

---

data.property

property

• `Const` **property**: typeof `property`

Unique symbol used as property key by the Property decorator to register decorated Model fields for further computation, e.g., serialization, treemapping etc.

See

Property

Defined in packages/data/src/relation/property.ts:24

Module: shell

shell

• **shell**: `Object`

`@sgrud/shell` - The SGRUD Web UI Shell.

The functions and classes found within the **`@sgrud/shell`** module are intended to ease the implementation of Component-based frontends by providing JSX runtime bindings via the `@sgrud/shell/jsx-runtime` module for the incremental-dom library and the Router to enable routing through Components based upon the SGRUD client libraries, but not limited to those. Furthermore, complex routing strategies and actions may be implemented through the interceptor-like Queue pattern.

Defined in packages/shell/index.ts:1

---

shell.Attribute

Attribute

▸ **Attribute**(`name?`): (`prototype`: `Component`, `propertyKey`: `PropertyKey`) => `void`

Component prototype property decorator factory. Applying the **Attribute** decorator to a property of a Component binds the decorated property to the corresponding **Attribute** of the respective Component. This implies that the **Attribute** `name` is appended to the observedAttributes array of the Component and the decorated property is replaced with a getter and setter deferring those operations to the **Attribute**. If no `name` supplied, the name of the decorated property will be used instead. Further, if both, a parameter initializer and an initial **Attribute** value are supplied, the **Attribute** value takes precedence.

`Example`

Bind a property to an **Attribute**:

```
import { Attribute, Component } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Attribute()
  public field?: string;

  public get template(): JSX.Element {
    return <span>Attribute value: {this.field}</span>;
  }

}
```

`See`

Component

Parameters

Name	Type	Description
`name?`	`string`	The Component **Attribute** `name`.

Returns `fn`

A Component prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`Component`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/shell/src/component/attribute.ts:45

---

shell.Catch

Catch

T **Catch**: (`error`: `unknown`) => `boolean` | `undefined`

The **Catch** type alias is used and intended to be used in conjunction with the CatchQueue and represents a function that is called with the thrown `error`. The return value of this callback will be used to examine wether the Component containing the decorated property is responsible to handle the thrown `error`.

See

CatchQueue

Defined in packages/shell/src/queue/catch.ts:61

packages/shell/src/queue/catch.ts:17

---

shell.Catch

Catch

▸ **Catch**(`trap?`): (`prototype`: `Component`, `propertyKey`: `PropertyKey`) => `void`

Component prototype property decorator factory. Applying the **Catch** decorator to a property, while optionally supplying a **trap** will navigate to the Component containing the decorated property when an error, **trap**ed by this **Catch** decorator, occurs during navigation.

Example

Catch all URIErrors:

```
import { Component, Catch } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Catch((error) => error instanceof URIError)
  public readonly error?: URIError;

  public get template(): JSX.Element {
    return <span>Error message: {this.error?.message}</span>;
  }

}
```

See

CatchQueue

Parameters

Name	Type	Description
`trap?`	`Catch`	The Catch callback to decide wether to `trap` an error.

Returns `fn`

A Component prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`Component`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/shell/src/queue/catch.ts:61

---

shell.CatchQueue

CatchQueue

• **CatchQueue**: `Object`

This built-in **CatchQueue** extension of the Queue base class is used by the Catch decorator to intercept Router navigation events and handles all errors thrown during the asynchronous evaluation of navigate invocations. When the Catch decorator is applied at least once this **CatchQueue** will be automatically provided as Queue to the Linker-

`Decorator`

Singleton

`See`

Queue

Defined in packages/shell/src/queue/catch.ts:115

---

shell.CatchQueue.[provide]

[provide][]

■ `Static` `Readonly` **[provide]**: `"sgrud.shell.Queue"`

Magic string by which this class is provided.

`See`

provide

Inherited from Queue.[provide]

Defined in packages/shell/src/queue/queue.ts:49

---

shell.CatchQueue.constructor

constructor

• **new CatchQueue**()

Public Singleton **constructor**. Called by the Catch decorator to link this Queue into the Router and to access the trapped and traps properties.

Overrides Queue.constructor

Defined in packages/shell/src/queue/catch.ts:143

---

shell.CatchQueue.handle

handle

▸ **handle**(_prev, next, queue): Observable<State<string>>

Overridden **handle** method of the Queue base class. Iterates all Segments of the next State and collects all traps for any encountered Components in those iterated Segments.

Parameters

Name	Type	Description
_prev	State<string>	The _previously active State (ignored).
next	State<string>	The next State navigated to.
queue	Queue	The next Queue to **handle** the navigation.

Returns Observable<State<string>>

An Observable of the **handle**d State.

Overrides Queue.handle

Defined in packages/shell/src/queue/catch.ts:161

---

shell.CatchQueue.trapped

trapped

• Readonly **trapped**: Map<Function, Record<PropertyKey, unknown>>

Mapping of all decorated Components to a Map of property keys and **trapped** errors.

Defined in packages/shell/src/queue/catch.ts:122

---

shell.CatchQueue.traps

traps

• Readonly **traps**: Map<Function, Map<PropertyKey, Catch>>

Mapping of all decorated Components to a Map of property keys and their **traps**.

Defined in packages/shell/src/queue/catch.ts:128

---

shell.CatchQueue.handleErrors

handleErrors

▸ `Private` **handleErrors**(): `Observable<never>`

handleErrors helper method returning an Observable from the global `window.onerror` and `window.unhandledrejection` event emitters. The returned Observable will either NEVER complete or invoke throwError with any globally emitted ErrorEvent or the reason for a PromiseRejectionEvent while subscribed to.

`Throws`

An Observable of any globally emitted error or rejection.

Returns `Observable<never>`

An Observable that NEVER completes.

Defined in packages/shell/src/queue/catch.ts:258

---

shell.CatchQueue.router

router

• `Private Readonly` **router**: `Router`

Factored-in **router** property linking the Router.

`Decorator`

Factor

Defined in packages/shell/src/queue/catch.ts:136

---

shell.Component

Component

▸ **Component**<`S`, `K`>(`selector`, `inherits?`): <`T`>(`constructor`: `T`) => `T`

Class decorator factory. Registers the decorated class as **Component** through the customElements registry. Registered **Component**s can be used in conjunction with any of the Attribute, Fluctuate and Reference prototype property decorators which will trigger their respective callbacks or renderComponent whenever one of the observedAttributes, observedFluctuations or observedReferences changes. While any **Component** registered by this decorator is enriched with basic rendering functionality, any implemented method will cancel out its `super` logic.

`Example`

Register a **Component**:

```
import { Component } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  public readonly styles: string[] = [`
    span {
      font-style: italic;
    }
  `];

  public get template(): JSX.Element {
    return <span>Example component</span>;
  }
```

}

See

- Attribute
- Reference

Type parameters

Name	Type	Description
`S`	extends `CustomElementTagName`	The custom **Component** tag name `selector` type.
`K`	extends `HTMLElementTagName`	-

Parameters

Name	Type	Description
`selector`	`S`	The custom **Component** tag name `selector`.
`inherits?`	`K`	The HTMLElement this **Component** `inherits` from.

Returns `fn`

A class constructor decorator.

▸ `<T>(constructor): T`

Type parameters

Name	Type
`T`	extends () => `Component` & `HTMLElementTagNameMap[S]` & `HTMLElementTagNameMap[K]`

Parameters

Name	Type
`constructor`	`T`

Returns `T`

Defined in packages/shell/src/component/component.ts:154

---

shell.Component

Component

• **Component**: `Object`

An interface describing the shape of a **Component**. Mostly adheres to the Web Components specification while providing rendering and change detection capabilities.

Defined in packages/shell/src/component/component.ts:154

packages/shell/src/component/component.ts:16

---

shell.Component.adoptedCallback

adoptedCallback

▸ `Optional` **adoptedCallback**(): `void`

Called when the Component is moved between Documents.

Returns `void`

Defined in packages/shell/src/component/component.ts:52

---

shell.Component.attributeChangedCallback

attributeChangedCallback

▸ `Optional` **attributeChangedCallback**(`name`, `prev?`, `next?`): `void`

Called when one of the Component's observed Attributes is added, removed or changed. Which Component attributes are observed depends on the contents of the observedAttributes array.

Parameters

Name	Type	Description
`name`	`string`	The `name` of the changed attribute.
`prev?`	`string`	The `previous` value of the changed attribute.
`next?`	`string`	The `next` value of the changed attribute.

Returns `void`

Defined in packages/shell/src/component/component.ts:63

---

shell.Component.connectedCallback

connectedCallback

▸ `Optional` **connectedCallback**(): `void`

Called when the Component is appended to the Document.

Returns `void`

Defined in packages/shell/src/component/component.ts:68

---

shell.Component.constructor

constructor

• **constructor**: `Object`

Inherited from HTMLElement.constructor

---

shell.Component.disconnectedCallback

disconnectedCallback

▸ `Optional` **disconnectedCallback**(): `void`

Called when the Component is removed from the Document.

Returns `void`

Defined in packages/shell/src/component/component.ts:73

---

shell.Component.fluctuationChangedCallback

fluctuationChangedCallback

▸ `Optional` **fluctuationChangedCallback**(`propertyKey`, `prev`, `next`): `void`

This callback is invoked whenever a Component Fluctuates, i.e., if the any of its decorated `propertyKeys` is assigned the `next` value emitted by one of the observedFluctuations.

Parameters

Name	Type	Description
`propertyKey`	`PropertyKey`	The `propertyKey` that Fluctuated.
`prev`	`unknown`	-
`next`	`unknown`	The `previous` value of the Fluctuated `propertyKey`.

Returns `void`

Defined in packages/shell/src/component/component.ts:84

---

shell.Component.observedAttributes

observedAttributes

• `Optional` `Readonly` **observedAttributes**: `string`[]

Array of Attribute names, which should be observed for changes, which will trigger the attributeChanged-Callback.

Defined in packages/shell/src/component/component.ts:22

---

shell.Component.observedFluctuations

observedFluctuations

• `Optional` `Readonly` **observedFluctuations**: `Record<PropertyKey, Subscription>`

A Record of Subscriptions opened by the Fluctuate decorator which trigger the fluctuationChangedCall-back upon each emission, while subscribed to.

Defined in packages/shell/src/component/component.ts:29

---

shell.Component.observedReferences

observedReferences

• `Optional` `Readonly` **observedReferences**: `Record<Key, keyof HTMLElementEventMap[]>`

A Record of References and observed events, which, when emitted by the reference, trigger the refer-enceChangedCallback.

Defined in packages/shell/src/component/component.ts:35

---

shell.Component.referenceChangedCallback

referenceChangedCallback

▸ `Optional` **referenceChangedCallback**(`key`, `node`, `event`): `void`

Called when one of the Component's Referenced and observed nodes emits an event. Which Referenced nodes are observed for which events depends on the contents of the observedReferences mapping.

Parameters

Name	Type	Description
`key`	`Key`	The `key` used to Reference the `node`.
`node`	`Node`	The Referenced `node`.
`event`	`Event`	The `event` emitted by the `node`.

Returns `void`

Defined in packages/shell/src/component/component.ts:99

---

shell.Component.renderComponent

renderComponent

▸ `Optional` **renderComponent**(): `void`

Called when the Component has changed and should render.

Returns `void`

Defined in packages/shell/src/component/component.ts:104

---

shell.Component.styles

styles

• `Optional` `Readonly` **styles**: `string`[]

Array of CSS **styles** in string form, which should be included within the ShadowRoot of the Component.

Defined in packages/shell/src/component/component.ts:41

---

shell.Component.template

template

• `Optional` `Readonly` **template**: `Element`

JSX representation of the Component **template**. If no template is supplied, an HTMLSlotElement will be rendered instead.

Defined in packages/shell/src/component/component.ts:47

---

shell.CustomElementTagName

CustomElementTagName

T **CustomElementTagName**: `Extract<keyof HTMLElementTagNameMap, '${string}-${string}'>`

String literal helper type. Enforces any assigned string to be a `keyof` HTMLElementTagNameMap, while excluding built-in tag names, i.e., extracting `${string}-${string}` keys of the HTMLElementTagNameMap.

`Example`

A valid **CustomElementTagName**:

```
const tagName: CustomElementTagName = 'example-component';
```

Defined in

packages/shell/src/component/runtime.ts:18

---

shell.Fluctuate

Fluctuate

▸ **Fluctuate**(`streamFactory`): (`prototype`: `Component`, `propertyKey`: `PropertyKey`) => `void`

Component prototype property decorator factory. Applying this **Fluctuate** decorator to a property of a custom Component while supplying a `streamFactory` that returns an ObservableInput upon invocation will subscribe the fluctuationChangedCallback method to each emission from this ObservableInput and replace the decorated property with a getter returning its last emitted value. Further, the resulting subscription, referenced by the decorated property, is assigned to the observedFluctuations property and may be terminated by unsubscribing manually. Finally, the Component will seize to **Fluctuate** automatically when it's disconnected from the Document.

`Example`

A Component that **Fluctuate**s:

```
import { Component, Fluctuate } from '@sgrud/shell';
import { fromEvent } from 'rxjs';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Fluctuate(() => fromEvent(document, 'click'))
  private readonly pointer?: MouseEvent;

  public get template(): JSX.Element {
    return <span>Clicked at ({this.pointer?.x}, {this.pointer?.y})</span>;
  }

}
```

`See`

Component

Parameters

Name	Type	Description
`streamFactory`	() => `ObservableInput<unknown>`	A forward reference to an ObservableInput.

Returns `fn`

A Component prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`Component`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/shell/src/component/fluctuate.ts:47

---

shell.HTMLElementTagName

HTMLElementTagName

T **HTMLElementTagName**: `Exclude`<keyof `HTMLElementTagNameMap`, '`${string}-${string}`'>

String literal helper type. Enforces any assigned string to be a `keyof` HTMLElementTagNameMap, while excluding custom element tag names, i.e., `${string}-${string}` keys of the HTMLElementTagNameMap.

`Example`

A valid **HTMLElementTagName**:

```
const tagName: HTMLElementTagName = 'div';
```

Defined in packages/shell/src/component/runtime.ts:32

---

shell.JSX

JSX

• **JSX**: `Object`

The intrinsic JSX namespace used by TypeScript to determine the Element type and all valid IntrinsicElements.

Defined in packages/shell/src/component/runtime.ts:40

---

shell.JSX.Element

Element

T **Element**: () => `Node`[]

Intrinsic JSX **Element** type helper representing an array of bound elementOpen and elementClose calls.

Defined in packages/shell/src/component/runtime.ts:46

---

shell.JSX.IntrinsicElements

IntrinsicElements

T **IntrinsicElements**: { [K in keyof HTMLElementTagNameMap]: Partial<Assign<Object, HTMLElementTagNameMap[K]>> & Object }

List of known JSX **IntrinsicElements**, comprised of the global HTMLElementTagNameMap.

Defined in packages/shell/src/component/runtime.ts:52

---

shell.JSX.Key

Key

T **Key**: `string` | `number`

Key references type helper. Enforces any assigned values to be of a compatible **Key** type.

Defined in packages/shell/src/component/runtime.ts:84

---

shell.Queue

Queue

• `Abstract` **Queue**: `Object`

Abstract base class to implement Router **Queue**s. By applying the Target decorator or otherwise providing an implementation of this abstract **Queue** base class to the Linker, the implemented handle method is called whenever a new State is triggered by navigating. This interceptor-like pattern makes complex routing strategies like asynchronous module-retrieval and the similar tasks easy to be implemented.

`Decorator`

Provide

`Example`

Simple **Queue** stub:

```
import { Provider, Target } from '@sgrud/core';
import { type Router, type Queue } from '@sgrud/shell';
import { type Observable } from 'rxjs';

@Target()
export class ExampleQueue
  extends Provider<typeof Queue>('sgrud.shell.Queue') {

  public override handle(
    prev: Router.State,
    next: Router.State,
    queue: Router.Queue
  ): Observable<Router.State> {
    throw new Error('Stub!');
  }

}
```

`See`

- Route
- Router

Defined in packages/shell/src/queue/queue.ts:42

---

shell.Queue.[provide]

[provide][]

■ `Static` `Readonly` **[provide]**: `"sgrud.shell.Queue"`

Magic string by which this class is provided.

`See`

provide

Defined in packages/shell/src/queue/queue.ts:49

---

shell.Queue.constructor

constructor

• **new Queue**()

---

shell.Queue.handle

handle

▸ `Abstract` **handle**(`prev`, `next`, `queue`): `Observable<State<string>>`

Abstract **handle** method, called whenever a new State should be navigated to. This method provides the possibility to intercept these upcoming States and, e.g., mutate or redirect them, i.e., **handle** the navigation.

Parameters

Name	Type	Description
`prev`	`State<string>`	The previously active State.
`next`	`State<string>`	The next State navigated to.
`queue`	`Queue`	The next Queue to **handle** the navigation.

Returns `Observable<State<string>>`

An Observable of the **handle**d State.

Defined in packages/shell/src/queue/queue.ts:62

---

shell.Reference

Reference

▸ **Reference**(`reference`, `observe?`): (`prototype`: `Component`, `propertyKey`: `PropertyKey`) => `void`

Component prototype property decorator factory. Applying this **Reference** decorator to a property of a registered Component while supplying the `referenceing` Key] and, optionally, an array of event names to `observe`, will replace the decorated property with a getter returning the `referenced` node, once rendered. If an array of event names is supplied, whenever one of those `observed` events is emitted by the `referenced` node, the referenceChangedCallback of the Component is called with the `reference` key, the `referenced` node and the emitted event.

`Example`

Reference a node:

```
import { Component, Reference } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Reference('example-key')
  private readonly span?: HTMLSpanElement;

  public get template(): JSX.Element {
    return <span key="example-key"></span>;
  }

}
```

`See`

Component

Parameters

Name	Type	Description
`reference`	`Key`	The `referenceing` Key.
`observe?`	keyof `HTMLElementEventMap[]`	An array of event names to `observe`.

Returns `fn`

A Component prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`Component`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/shell/src/component/reference.ts:48

---

shell.Resolve

Resolve

T **Resolve**<`S`>: (`segment`: `Segment<S>`, `state`: `State<S>`) => `ObservableInput<unknown>`

Type parameters

Name	Type	Description
`S`	extends `string`	The Route path string type.

Type declaration ▸ (`segment`, `state`): `ObservableInput<unknown>`

The **Resolve** type alias is used and intended to be used in conjunction with the ResolveQueue Queue and the Resolve decorator. The **Resolve** type alias represents a function that will be called with the respective Segment and State.

`See`

Resolve

Parameters

Name	Type
`segment`	`Segment<S>`
`state`	`State<S>`

Returns `ObservableInput<unknown>`

Defined in packages/shell/src/queue/resolve.ts:71

packages/shell/src/queue/resolve.ts:18

---

shell.Resolve

Resolve

▸ **Resolve**<S>(resolve): (prototype: Component, propertyKey: PropertyKey) => void

Component prototype property decorator factory. Applying the **Resolve** decorator to a property of a Component, while supplying an ObservableInput to be resolved, will replace the decorated property with a getter returning the **Resolve**d value the supplied ObservableInput resolves to. To do so the **Resolve** decorator relies on the built-in ResolveQueue.

Example

Resolve the Segment and State:

```
import { Component, Resolve } from '@sgrud/shell';
import { of } from 'rxjs';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Resolve((segment, state) => of([segment.route.path, state.search]))
  public readonly resolved!: [string, string];

  public get template(): JSX.Element {
    return <span>Resolved: {this.resolved.join('?')}</span>;
  }

}
```

See

ResolveQueue

Type parameters

Name	Type	Description
S	extends string	The Route path string type.

Parameters

Name	Type	Description
resolve	Resolve<S>	An ObservableInput to resolve.

Returns fn

A Component prototype property decorator.

▸ (prototype, propertyKey): void

Parameters

Name	Type
prototype	Component
propertyKey	PropertyKey

Returns void

Defined in packages/shell/src/queue/resolve.ts:71

---

shell.ResolveQueue

ResolveQueue

• **ResolveQueue**: `Object`

This built-in **ResolveQueue** extension of the Queue base class intercepts all navigational events of the Router to Resolve ObservableInputs before invoking subsequent Queues. Thereby this **ResolveQueue** allows asynchronous evaluations to be executed and their Resolved values to be provided to a Component, before it is rendered into a Document for the first time. When the Catch decorator is applied at least once this **ResolveQueue** will be automatically provided as Queue to the Linker.

`Decorator`

Singleton

`See`

Queue

Defined in packages/shell/src/queue/resolve.ts:127

---

shell.ResolveQueue.[provide]

[provide][]

■ `Static` `Readonly` **[provide]**: `"sgrud.shell.Queue"`

Magic string by which this class is provided.

`See`

provide

Inherited from Queue.[provide]

Defined in packages/shell/src/queue/queue.ts:49

---

shell.ResolveQueue.constructor

constructor

• **new ResolveQueue**()

Public Singleton **constructor**. Called by the Resolve decorator to link this Queue into the Router and to access the required and resolved properties.

Overrides Queue.constructor

Defined in packages/shell/src/queue/resolve.ts:147

---

shell.ResolveQueue.handle

handle

▸ **handle**(`_prev`, `next`, `queue`): `Observable<State<string>>`

Overridden **handle** method of the Queue base class. Iterates all Segments of the `next` State and collects all Resolvers for any encountered Components in those iterated Segments. The collected Resolvers are run before invoking the subsequent Queue.

Name	Type	Description

Parameters

Name	Type	Description
`_prev`	`State<string>`	The _previously active State (ignored).
`next`	`State<string>`	The `next` State navigated to.
`queue`	`Queue`	The next Queue to **handle** the navigation.

Returns `Observable<State<string>>`

An Observable of the **handle**d State.

Overrides Queue.handle

Defined in packages/shell/src/queue/resolve.ts:166

---

shell.ResolveQueue.required

required

• `Readonly` **required**: `Map<Function, Map<PropertyKey, Resolve<string>>>`

Mapping of all decorated Components to a Map of property keys and their **required** Resolvers.

Defined in packages/shell/src/queue/resolve.ts:134

---

shell.ResolveQueue.resolved

resolved

• `Readonly` **resolved**: `Map<Function, Record<PropertyKey, unknown>>`

Mapping of all decorated Components to an object consisting of property keys and their corresponding Resolved return values.

Defined in packages/shell/src/queue/resolve.ts:140

---

shell.Route

Route

▸ **Route**<S>(`config`): <T>(`constructor`: T) => `void`

Class decorator factory. Applying the **Route** decorator to a custom element will associate the supplied `config` with the decorated element constructor. Further, the `config`ured children are iterated over and every child that is a custom element itself will be replaced by its respective route configuration or ignored, if no configuration was associated with the child. Finally, the processed `config` is added to the Router.

`Example`

Associate a Route `config` to a Component:

```
import { Component, Route } from '@sgrud/shell';
import { ChildComponent } from './child-component';

@Route({
  path: 'example',
  children: [
    ChildComponent
```

```
  ]
})
@Component('example-element')
export class ExampleComponent extends HTMLElement implements Component {}
```

See

Router

Type parameters

Name	Type	Description
`S`	extends `string`	The Route path string type.

Parameters

Name	Type	Description
`config`	`Assign<{ children?: (Route<string> | CustomElementConstructor & { [route]?: Route<string> })[]; slots?: Record<string, CustomElementConstructor | CustomElementTagName> }, Omit<Route<S>, "component">> & { parent?: Route<string> | CustomElementConstructor & { [route]?: Route<string> } }`	The Route `config` for this element.

Returns `fn`

A class constructor decorator.

▸ `<T>(constructor): void`

Type parameters

Name	Type
`T`	extends `CustomElementConstructor & { [route]?: Route<S> }`

Parameters

Name	Type
`constructor`	`T`

Returns `void`

Defined in packages/shell/src/router/route.ts:94

---

shell.Route

Route

• **Route**<S>: `Object`

Interface describing the shape of a **Route**. A **Route** must consist of at least a path and may specify a component, as well as slots, which will be rendered into the RouterOutlet when the **Route** is navigated to. Furthermore a **Route** may also specify children.

`Example`

Define a **Route**:

```
import { type Route } from '@sgrud/shell';

const route: Route = {
  path: '',
  component: 'example-element',
  children: [
    {
      path: 'child',
      component: 'child-element'
    }
  ]
};
```

`See`

Router

Type parameters

Name	Type	Description
S	extends `string` = `string`	The **Route** path string type.

Defined in

packages/shell/src/router/route.ts:94

packages/shell/src/router/route.ts:33

---

shell.Route.children

children

• `Optional` `Readonly` **children**: `Route`<`string`>[]

Optional array of **children** for this Route.

Defined in

packages/shell/src/router/route.ts:38

---

shell.Route.component

component

• `Optional` `Readonly` **component**: `CustomElementTagName`

Optional Route **component**.

Defined in

packages/shell/src/router/route.ts:43

---

shell.Route.constructor

constructor

• **constructor**: `Object`

---

shell.Route.path

path

• `Readonly` **path**: `S`

Required Route **path**.

Defined in packages/shell/src/router/route.ts:48

---

shell.Route.slots

slots

• `Optional` `Readonly` **slots**: `Record<string, CustomElementTagName>`

Optional mapping of elements to their **slots**.

Defined in packages/shell/src/router/route.ts:53

---

shell.Router

Router

• **Router**: `Object`

Targeted Singleton **Router** class extending the built-in Set. This Singleton class provides routing and rendering capabilities. Routing is primarily realized by maintaining the inherited Set of Routes and (recursively) matching paths against those Routes, when instructed so by the navigate method. When a matching Segment is found, the corresponding Components are rendered by the handle method (which is part of the implemented Queue contract).

`Decorator`

Target

`Decorator`

Singleton

Defined in packages/shell/src/router/router.ts:13

packages/shell/src/router/router.ts:165

---

shell.Router.[observable]

[observable][]

▸ `Static` **[observable]**(): `Subscribable<Router>`

Static `Symbol.observable` method returning a Subscribable. The returned Subscribable mirrors the private loader and is used for initializations after a new global Route tree was added to the Router.

`Example`

Subscribe to the Router:

```
import { Router } from '@sgrud/shell';
import { from } from 'rxjs';

from(Router).subscribe(console.log);
```

Returns `Subscribable<Router>`

A Subscribable emitting this Router.

Defined in packages/shell/src/router/router.ts:192

---

shell.Router.loader

loader

■ `Static` `Private` **loader**: `ReplaySubject<Router>`

Private static ReplaySubject used as the Router **loader**. This **loader** emits every time Routes are added, whilst the size being `0`, so either for the first time after construction or after the Router was cleared.

Defined in packages/shell/src/router/router.ts:173

---

shell.Router.[iterator]

[iterator][]

• `Readonly` **[iterator]**: `never`

`declared` well-known `Symbol.iterator` property. This declaration enforces correct typing when retrieving the Subscribable from the well-known `Symbol.observable` method by voiding the inherited `Symbol.iterator`.

Overrides Set.[iterator]

Defined in packages/shell/src/router/router.ts:208

---

shell.Router.[observable]

[observable][]

▸ **[observable]**(): `Subscribable<State<string>>`

Well-known `Symbol.observable` method returning a Subscribable. The returned Subscribable emits the current State and every time this changes.

`Example`

Subscribe to upcoming States:

```
import { Router } from '@sgrud/shell';
import { from } from 'rxjs';

from(new Router()).subscribe(console.log);
```

Returns `Subscribable<State<string>>`

A Subscribable emitting States.

Defined in packages/shell/src/router/router.ts:286

---

shell.Router.add

add

▸ **add**(`route`): `Router`

Overridden **add** method. Invoking this method while supplying a `route` will **add** the supplied `route` to the Router after deleting its child Routes from the Router, thereby ensuring that only root `routes` remain part of the Router.

Parameters

Name	Type	Description
`route`	`Route<string>`	The Route to **add** to the Router.

Returns `Router`

This instance of the Router.

Overrides Set.add

Defined in packages/shell/src/router/router.ts:299

---

shell.Router.baseHref

baseHref

• `Readonly` **baseHref**: `string`

An absolute **baseHref** for navigation.

`Default Value`

`'/'`

Defined in packages/shell/src/router/router.ts:215

---

shell.Router.connect

connect

▸ **connect**(`this`, `outlet?`, `baseHref?`, `hashBased?`): `void`

connecting helper method. Calling this method will **connect** a handler to the global `onpopstate` event, invoking navigate with the appropriate arguments. This method furthermore allows the properties baseHref, hashBased and outlet to be overridden. Invoking the **connect** method throws an error if called more than once, without invoking the disconnect method in between invocations.

`Throws`

A ReferenceError if already **connect**ed.

Parameters

Name	Type	Description
`this`	`Mutable<Router>`	The Mutable explicit polymorphic `this` parameter.
`outlet`	`Element` | `DocumentFragment`	The rendering `outlet` for Routes.
`baseHref`	`string`	An absolute `baseHref` for navigation.
`hashBased`	`boolean`	Wether to employ `hashBased` routing.

Returns `void`

Defined in packages/shell/src/router/router.ts:331

---

shell.Router.constructor

constructor

• **new Router**()

Public Singleton Router class **constructor**. This **constructor** is called once by the Target decorator and sets initial values on this instance. All subsequent calls will return the previously constructed Singleton instance of this class.

Overrides Set<Route>.constructor

Defined in packages/shell/src/router/router.ts:251

---

shell.Router.disconnect

disconnect

▸ **disconnect**(`this`): `void`

disconnecting helper method. Calling this method (after calling connect) will **disconnect** the previously connected handler from the global `onpopstate` event. Further, the arguments passed to connect are revoked, meaning the default values of the properties baseHref, hashBased and outlet are restored. Calling this method without previously connecting the Router throws an error.

`Throws`

A ReferenceError if already **disconnect**ed.

Parameters

Name	Type	Description
`this`	`Mutable<Router>`	The Mutable explicit polymorphic `this` parameter.

Returns `void`

Defined in packages/shell/src/router/router.ts:373

---

shell.Router.handle

handle

▸ **handle**(`state`, `action?`): `Observable<State<string>>`

Implementation of the **handle** method as required by the Queue interface contract. It is called internally by the navigate method after all Queues have been invoked. It is therefore considered the default or fallback Queue and handles the rendering of the supplied `state`.

Parameters

Name	Type	Default value	Description
`state`	`State<string>`	`undefined`	The next State to handle.
`action`	`Action`	`'push'`	The Action to apply to the History.

Returns `Observable<State<string>>`

An Observable of the handled State.

Implementation of Queue.handle

Defined in packages/shell/src/router/router.ts:396

---

shell.Router.hashBased

hashBased

• `Readonly` **hashBased**: `boolean`

Wether to employ **hashBased** routing.

`Default Value`

`false`

Defined in packages/shell/src/router/router.ts:222

---

shell.Router.join

join

▸ **join**(`segment`): `string`

Segment **join**ing helper. The supplied `segment` is converted to a string by spooling to its top-most parent and iterating through all children while concatenating every encountered path. If said path is an (optional) parameter, this portion of the returned string is replaced by the respective Params value.

Parameters

Name	Type	Description
`segment`	`Segment<string>`	The Segment to be **join**ed.

Returns `string`

The **join**ed Segment in string form.

Defined in packages/shell/src/router/router.ts:438

---

shell.Router.lookup

lookup

▸ **lookup**(`selector`, `routes?`): `undefined` | `string`

Lookup helper method. Calling this method while supplying a `selector` and optionally an array of `routes` to iterate will return the **lookup**ed Route path for the supplied `selector` or `undefined`, if it does not occur within at least one route. When multiple occurrences of the same `selector` exist, the Route path to its first occurrence is returned.

Parameters

Name	Type	Description
`selector`	`string`	The Component `selector` to **lookup**.
`routes`	`Route<string>[]`	An array of `routes` to use for **lookup**.

Returns `undefined` | `string`

The **lookup**ed Route path or `undefined`.

Defined in packages/shell/src/router/router.ts:472

---

shell.Router.match

match

▸ **match**(path, routes?): undefined | Segment<string>

Main Router **match**ing method. Calling this method while supplying a path and optionally an array of routes will return the first **match**ing Segment or undefined, if nothing **match**es. If no routes are supplied, routes previously added to the Router will be used. The **match** method represents the backbone of this Router class, as it, given a list of routes and a path, will determine wether this path represents a **match** within the list of routes, thereby effectively determining navigational integrity.

Example

Test if path 'example/route' **match**es child or route:

```
import { Router } from '@sgrud/shell';

const path = 'example/route';
const router = new Router();

const child = {
  path: 'route'
};

const route = {
  path: 'example',
  children: [child]
};

router.match(path, [child]); // false
router.match(path, [route]); // true
```

Parameters

Name	Type	Description
path	string	The path to **match** routes against.
routes	Route<string>[]	An array of routes to use for **match**ing.

Returns undefined | Segment<string>

The first **match**ing Segment or undefined.

Defined in packages/shell/src/router/router.ts:526

---

shell.Router.navigate

navigate

▸ **navigate**(target, search?, action?): Observable<State<string>>

Main **navigate** method. Calling this method while supplying either a path or Segment as navigation target and optional search parameters will normalize the supplied path by trying to match a respective Segment or directly use the supplied Segment for the next State. This upcoming State is looped through all linked Queues and finally handled by the Router itself to render the resulting, possibly intercepted and mutated State.

Throws

An Observable URIError, if nothing matches.

Parameters

Name	Type	Default value	Description
`target`	`string` | `Segment<string>`	`undefined`	Path or Segment to **navigate** to.
`search?`	`string`	`undefined`	Optional `search` parameters in string form.
`action`	`Action`	`'push'`	The Action to apply to the History.

Returns `Observable<State<string>>`

An Observable of the **navigate**d State.

Defined in packages/shell/src/router/router.ts:620

---

shell.Router.outlet

outlet

• `Readonly` **outlet**: `Element` | `DocumentFragment`

The rendering **outlet** for navigated Routes.

`Default Value`

`document.body`

Defined in packages/shell/src/router/router.ts:229

---

shell.Router.rebase

rebase

▸ **rebase**(`path`, `prefix?`): `string`

rebase helper method. **rebase**s the supplied `path` against the current baseHref, by either `prefix`ing the baseHref to the supplied `path` or stripping it, depending on the `prefix` argument.

Parameters

Name	Type	Default value	Description
`path`	`string`	`undefined`	The `path` to **rebase** against the baseHref.
`prefix`	`boolean`	`true`	Wether to `prefix` or strip the baseHref.

Returns `string`

The `path` **rebase**d against the baseHref.

Defined in packages/shell/src/router/router.ts:679

---

shell.Router.spool

spool

▸ **spool**(`segment`, `rewind?`): `Segment<string>`

spooling helper method. Given a `segment` (and wether to `rewind`), the top-most parent (or deepest child) of the graph-link Segment is returned.

Parameters

Name	Type	Default value	Description
`segment`	`Segment<string>`	`undefined`	The Segment to **spool**.
`rewind`	`boolean`	`true`	Wether to `rewind` the **spool** direction.

Returns `Segment<string>`

The **spool**ed Segment.

Defined in packages/shell/src/router/router.ts:705

---

shell.Router.state

state

• `get` **state**(): `State<string>`

Getter mirroring the current value of the internal changes BehaviorSubject.

Returns `State<string>`

Defined in packages/shell/src/router/router.ts:241

---

shell.Router.changes

changes

• `Private` `Readonly` **changes**: `BehaviorSubject<State<string>>`

Internally used BehaviorSubject containing and emitting every navigated State.

Defined in packages/shell/src/router/router.ts:235

---

shell.Router

Router

• **Router**: `Object`

Namespace containing types and interfaces used and intended to be used in conjunction with the Singleton Router class.

See

Router

Defined in packages/shell/src/router/router.ts:13

packages/shell/src/router/router.ts:165

---

shell.Router.Action

Action

T **Action**: `"pop"` | `"push"` | `"replace"`

Type alias constraining the possible Router **Action**s to `'pop'`, `'push'` and `'replace'`. These **Action**s correspond loosely to possible History events.

Defined in packages/shell/src/router/router.ts:20

---

shell.Router.Left

Left

T **Left**<`S`>: `S` extends `${infer I}/${string}` ? `I` : `S`

String literal helper type. Represents the **Left**est part of a Route path.

Example

Left of `'nested/route/path'`:

```ts
import { type Router } from '@sgrud/shell';

const left: Router.Left<'nested/route/path'>; // 'nested'
```

Type parameters

Name	Type	Description
`S`	extends `string`	The Route path string type.

Defined in packages/shell/src/router/router.ts:36

---

shell.Router.Params

Params

T **Params**<`S`>: `S` extends `${string}:${infer P}` ? `P` extends `${Left<P>}${infer I}` ? `Params<I>` : `never` & `Left<P>` extends `${infer I}?` ? { [K in I]?: string } : { [K in Left<P>]: string } : {}

Type helper representing the (optional) **Params** of a Route path. By extracting string literals starting with a colon (and optionally ending on a question mark), a union type of a key/value pair for each parameter is created.

Example

Extract **Params** from `'item/:id/field/:name?'`:

```ts
import { type Router } from '@sgrud/shell';

const params: Router.Params<'item/:id/field/:name?'>;
// { id: string; name?: string; }
```

Type parameters

Name	Description
`S`	The Route path string type.

Defined in packages/shell/src/router/router.ts:55

---

shell.Router.Queue

Queue

• **Queue**: `Object`

Interface describing the shape of a **Queue**. These **Queue**s are run whenever a navigation is triggered and may intercept and mutate the next State or completely block or redirect a navigation.

See

Queue

Defined in packages/shell/src/router/router.ts:72

Router.Queue.handle

handle

▸ **handle**(`next`): `Observable<State<string>>`

handle method, called when a navigation was triggered.

Parameters

Name	Type	Description
`next`	`State<string>`	The `next` State to be **handle**d.

Returns `Observable<State<string>>`

An Observable of the **handle**d State.

Defined in packages/shell/src/router/router.ts:80

---

shell.Router.Segment

Segment

• **Segment**<`S`>: `Object`

Interface describing the shape of a Router **Segment**. A **Segment** represents a navigated Route and its corresponding Params. As Routes are represented in a tree-like structure and one **Segment** represents one layer within the Route-tree, each **Segment** may have a parent and/or a child. The resulting graph of **Segment**s represents the navigated path through the underlying Route-tree.

Type parameters

Name	Type	Description
`S`	extends `string` = `string`	The Route path string type.

Defined in packages/shell/src/router/router.ts:95

Router.Segment.child

child

• `Optional` `Readonly` **child**: `Segment<string>`

Optional **child** of this Segment.

Defined in packages/shell/src/router/router.ts:100

Router.Segment.params

params

• `Readonly` **params**: `Params<S>`

Route path Params and their corresponding values.

Defined in packages/shell/src/router/router.ts:105

Router.Segment.parent

parent

• `Optional` `Readonly` **parent**: `Segment<string>`

Optional **parent** of this Segment.

Defined in packages/shell/src/router/router.ts:110

Router.Segment.route

route

• `Readonly` **route**: `Route<S>`

Route associated with this Segment.

Defined in packages/shell/src/router/router.ts:115

---

shell.Router.State

State

• **State**<`S`>: `Object`

Interface describing the shape of a **State** of the Router. **State**s correspond to the History, as each navigation results in a new **State** being created. Each navigated **State** is represented by its absolute path its search parameters and a segment as entrypoint to the graph-like representation of the navigated path through the route-tree.

Type parameters

Name	Type	Description
`S`	extends `string` = `string`	The Route path string type.

Defined in packages/shell/src/router/router.ts:129

Router.State.path

path

• `Readonly` **path**: `S`

Absolute **path** of the State.

Defined in packages/shell/src/router/router.ts:134

Router.State.search

search

• `Readonly` **search**: `string`

search parameters of the State.

Defined in packages/shell/src/router/router.ts:139

Router.State.segment

segment

• `Readonly` **segment**: `Segment<S>`

Segment of the State.

Defined in packages/shell/src/router/router.ts:144

---

shell.RouterLink

RouterLink

• **RouterLink**: `Object`

Custom element extending the HTMLAnchorElement. This element provides a declarative way to invoke the navigate method within the bounds of the RouterOutlet, while maintaining compatibility with SSR/SEO aspects of SPAs. This is achieved by rewriting its href against the baseHref and intercepting the default browser behavior when onclicked.

Example

A `router-link`:

```
<a href="/example" is="router-link">Example</a>
```

See

Router

Defined in packages/shell/src/router/link.ts:32

---

shell.RouterLink.observedAttributes

observedAttributes

■ `Static` `Readonly` **observedAttributes**: `string[]`

Array of attribute names that should be observed for changes, which will trigger the attributeChanged-Callback. This element only observes its href attribute.

Defined in packages/shell/src/router/link.ts:39

---

shell.RouterLink.attributeChangedCallback

attributeChangedCallback

▸ **attributeChangedCallback**(`_name`, `_prev?`, `next?`): `void`

This method is called whenever this element's href attribute is added, removed or changed. The `next` attribute value is used to determine wether to rebase the href.

Parameters

Name	Type	Description
`_name`	`string`	The `_name` of the changed attribute (ignored).
`_prev?`	`string`	The `_previous` value of the changed attribute (ignored).
`next?`	`string`	The `next` value of the changed attribute.

Returns `void`

Defined in packages/shell/src/router/link.ts:75

---

shell.RouterLink.constructor

constructor

• **new RouterLink**()

Public **constructor** of this custom RouterLink element. This **constructor** is called whenever a new instance this custom element is being rendered into a Document.

Overrides HTMLAnchorElement.constructor

Defined in packages/shell/src/router/link.ts:56

---

shell.RouterLink.onclick

onclick

• `Readonly` **onclick**: (`event`: `MouseEvent`) => `void`

Type declaration ▸ (`event`): `void`

Overridden **onclick** handler, preventing the default browser behavior and invoking navigate instead.

Parameters

Name	Type	Description
`event`	`MouseEvent`	The **onclick** fired MouseEvent.

Returns `void`

Overrides HTMLAnchorElement.onclick

Defined in packages/shell/src/router/link.ts:92

---

shell.RouterLink.router

router

• `Private` `Readonly` **router**: `Router`

Factored-in **router** property linking the Router.

`Decorator`

Factor

Defined in packages/shell/src/router/link.ts:49

---

shell.RouterOutlet

RouterOutlet

• **RouterOutlet**: `Object`

Custom element extending the HTMLSlotElement. When this element is constructed, it supplies the value of its baseHref attribute and the presence of a hashBased attribute on itself to the Router while connecting the Router to itself. This element should only be used once, as it will be used by the Router as outlet to render the current State.

`Example`

A `router-outlet`:

```
<slot baseHref="/example" is="router-outlet">Loading...</slot>
```

`See`

Router

Defined in

packages/shell/src/router/outlet.ts:33

---

shell.RouterOutlet.baseHref

baseHref

• `get` **baseHref**(): `undefined` | `string`

Getter mirroring the **baseHref** attribute of this element.

Returns

`undefined` | `string`

Defined in

packages/shell/src/router/outlet.ts:46

---

shell.RouterOutlet.constructor

constructor

• **new RouterOutlet**()

Public **constructor** of this custom RouterOutlet element. Supplies the value of its baseHref attribute and the presence of a hashBased attribute on itself to the Router while connecting the Router to itself.

Overrides

HTMLSlotElement.constructor

Defined in

packages/shell/src/router/outlet.ts:63

---

shell.RouterOutlet.hashBased

hashBased

• `get` **hashBased**(): `boolean`

Getter mirroring the presence of a **hashBased** attribute on this element.

Returns

`boolean`

Defined in

packages/shell/src/router/outlet.ts:53

---

shell.RouterOutlet.router

router

• `Private Readonly` **router**: `Router`

Factored-in **router** property linking the Router.

`Decorator`

Factor

Defined in packages/shell/src/router/outlet.ts:41

---

shell.component

component

• `Const` **component**: typeof `component`

Unique symbol used as property key by the Component decorator to associate the supplied constructor with its wrapper.

Defined in packages/shell/src/component/component.ts:9

---

shell.createElement

createElement

▸ **createElement**(`type`, `props?`, `ref?`): `Element`

Element factory. Provides JSX runtime compliant bindings creating arrays of bound elementOpen and elementClose calls. This **createElement** factory function is meant to be implicitly imported by the TypeScript transpiler through its JSX bindings and returns an array of bound elementOpen and elementClose function calls, representing the created Element. This array of bound functions can be rendered into an element attached to the Document through the render function.

`See`

render

Parameters

Name	Type	Description
`type`	`Function` | keyof `HTMLElementTagNameMap`	The `type` of Element to create.
`props?`	`Record<string, any>`	Any properties to assign to the created Element.
`ref?`	`Key`	An optional `reference` to the created Element.

Returns `Element`

An array of bound functions representing the Element.

Defined in packages/shell/src/component/runtime.ts:116

---

shell.createFragment

createFragment

▸ **createFragment**(`props?`): `Element`

JSX fragment factory. Provides a JSX runtime compliant helper creating arrays of bound elementOpen and elementClose calls. This **createFragment** factory function is meant to be implicitly imported by the TypeScript transpiler through its JSX bindings and returns an Element which can be rendered into an element attached to the Document through the render function.

Parameters

Name	Type	Description
`props?`	`Record<string, any>`	Any properties to assign to the created Element.

Returns `Element`

An array of bound functions representing the Element.

Defined in packages/shell/src/component/runtime.ts:179

---

shell.customElements

customElements

• `Const` **customElements**: `CustomElementRegistry` & { `getName`: (`constructor`: `CustomElementConstructor`) => `undefined` | `string` }

Proxy around the built-in CustomElementRegistry, maintaining a mapping of all registered elements and their corresponding names, which can be queried by calling getName.

`Remarks`

https://github.com/WICG/webcomponents/issues/566

Defined in packages/shell/src/component/registry.ts:13

---

shell.html

html

▸ **html**(`contents`, `ref?`): `Element`

Raw **html** rendering helper function. As JSX is pre-processed by the TypeScript transpiler, assigning directly to the `innerHTML` property of an Element will not result in the `innerHTML` to be rendered in the Element. To insert raw **html** into an Element this helper function has to be employed.

Parameters

Name	Type	Description
`contents`	`string`	The raw **html** `contents` to render.
`ref?`	`Key`	An optional `reference` to the created Element.

Returns `Element`

An array of bound functions representing the Element.

Defined in packages/shell/src/component/runtime.ts:205

---

shell.references

references

▸ **references**(`outlet`): `Map<Key, Node>` | `undefined`

JSX **references** helper. Calling this function while supplying a viable `outlet` will return all referenced Elements mapped by their corresponding Keys known to the supplied `outlet`. A viable `outlet` may be any element which previously was passed as `outlet` to the render function.

Parameters

Name	Type	Description
`outlet`	`Element` | `DocumentFragment`	The `outlet` to return **references** for.

Returns `Map<Key, Node>` | `undefined`

Any **references** known to the supplied `outlet`.

Defined in packages/shell/src/component/runtime.ts:226

---

shell.render

render

▸ **render**(`outlet`, `element`): `Node`

JSX **render**ing helper. This helper is a small wrapper around the patch function and **render**s a Element created through the createElement factory into the supplied `outlet`.

See

createElement

Parameters

Name	Type	Description
`outlet`	`Element` | `DocumentFragment`	The `outlet` to **render** the `element` into.
`element`	`Element`	JSX `element` to be **render**ed.

Returns `Node`

Rendered `outlet` element.

Defined in packages/shell/src/component/runtime.ts:243

---

shell.route

route

• `Const` **route**: typeof `route`

Unique symbol used as property key by the Route decorator to associate the supplied Route configuration with the decorated element.

Defined in packages/shell/src/router/route.ts:62

Module: state

state

• **state**: `Object`

`@sgrud/state` - The SGRUD State Machine.

The functions and classes found within the **`@sgrud/state`** module are intended to ease the implementation of Stateful data Stores within applications built upon the SGRUD client libraries. Through wrappers

around the IndexedDB and SQLite3 storage Drivers, data will be persisted in every environment. Furthermore, through the employment of Effects, side-effects like retrieving data from external services or dispatching subsequent Actions can be easily achieved.

The **`@sgrud/state`** module includes a standalone JavaScript bundle which is used to fork a background Thread upon import of this module. This background Thread is henceforth used for State mutation and persistance, independently of the foreground process. Depending on the runtime environment, either a `navigator.serviceWorker` is `registered` or a `new require('worker_threads').Worker()` NodeJS equivalent will be forked.

Defined in packages/state/index.ts:1

---

state.DispatchEffect

DispatchEffect

• **DispatchEffect**: `Object`

Built-in **DispatchEffect** extending the abstract Effect base class. This **DispatchEffect** is automatically implanted when the `@sgrud/state` module is imported and can therefore be always used in Actions.

`Decorator`

Implant

`See`

Effect

Defined in packages/state/src/effect/dispatch.ts:68

---

state.DispatchEffect.constructor

constructor

• **new DispatchEffect**()

Public **constructor** (which should never be called).

`Throws`

A TypeError upon construction.

Inherited from Effect.constructor

Defined in packages/state/src/effect/effect.ts:71

---

state.DispatchEffect.function

function

▸ **function**(`this`): <T>(handle: `${string}.${string}.${string}`, ...action: `Action<T>`) => `Promise<State<T>>`

Overridden **function** binding the DispatchEffect to the polymorphic `this` of the StateWorker.

Parameters

Name	Type	Description
`this`	`StateWorker`	The explicit polymorphic `this` parameter.

Returns `fn`

This DispatchEffect bound to the StateWorker.

▸ `<T>(handle, ...action): Promise<State<T>>`

Implanted DispatchEffect providing a convenient way to **dispatch** Actions from within Actions. This DispatchEffect in combination with the StateEffect can be used to implement complex interactions between different Stores.

`Example`

dispatch an Action to another Store:

```
import { type Bus } from '@sgrud/bus';
import { Stateful, Store } from '@sgrud/state';

@Stateful('io.github.sgrud.store.example', { state: undefined })
export class ExampleStore extends Store<ExampleStore> {

  public readonly state?: Store.State<Store>;

  public async dispatchAction<T extends Store>(
    handle: Bus.Handle,
    ...action: Store.Action<T>
  ): Promise<Store.State<this>> {
    const state = await sgrud.state.effects.dispatch<T>(
      handle, ...action
    );

    return { ...this, state };
  }

}
```

`See`

DispatchEffect

Type parameters

Name	Type	Description
`T`	extends `Store<any, T>`	The extending Store InstanceType.

Parameters

Name	Type	Description
`handle`	`` `${string}.${string}.${string}` ``	The Handle representing the Store.
`...action`	`Action<T>`	A type-guarded Action to **dispatch**.

Returns `Promise<State<T>>`

The next State after **dispatch**ing.

Overrides Effect.function

Defined in packages/state/src/effect/dispatch.ts:77

---

state.Effect

Effect

• `Abstract` **Effect**<`K`>: `Object`

Abstract **Effect** base class. When this class is extended and decorated with the Implant decorator or implanted through the StateHandler, its function will be made available to Actions through the global sgrud.state.effects namespace.

`Example`

An `importScripts` **Effect**:

```ts
import { Effect, Implant, type StateWorker, type Store } from '@sgrud/state';

declare global {
  namespace sgrud.state.effects {
    function importScripts(...urls: (string | URL)[]): Promise<void>;
  }
}

@Implant('importScripts')
export class FetchEffect extends Effect {

  public override function(
    this: StateWorker
  ): Store.Effects['importScripts'] {
    return async(...urls) => {
      return importScripts(...urls);
    };
  }

}
```

Type parameters

Name	Type	Description
`K`	extends `Effect` = `Effect`	The Effect `locate` type.

Defined in

packages/state/src/effect/effect.ts:64

---

state.Effect.constructor

constructor

• **new Effect**<`K`>()

Public **constructor** (which should never be called).

`Throws`

A TypeError upon construction.

Type parameters

Name	Type
`K`	extends `"fetch"` | `"state"` | `"dispatch"` = `"fetch"` | `"state"` | `"dispatch"`

Defined in

packages/state/src/effect/effect.ts:71

---

state.Effect.function

function

▸ `Abstract` **function**(`this`): typeof `effects`[`K`]

Abstract **function** responsible for returning the bound Effect. When an implanted Effect is invoked, it is bound to the polymorphic `this` of the StateWorker upon invocation. This **function** provides the means of interacting with this bond, as in, utilizing the polymorphic `this` of the StateWorker to provide the bound Effect, e.g., by utilizing `protected` properties and methods of the bound-to StateWorker.

Parameters

Name	Type	Description
`this`	`StateWorker`	The explicit polymorphic `this` parameter.

Returns typeof `effects`[`K`]

This Effect bound to the StateWorker.

Defined in packages/state/src/effect/effect.ts:87

---

state.FetchEffect

FetchEffect

• **FetchEffect**: `Object`

Built-in **FetchEffect** extending the abstract Effect base class. This **FetchEffect** is automatically implanted when the `@sgrud/state` module is imported and can therefore be always used in Actions.

`Decorator`

Implant

`See`

Effect

Defined in packages/state/src/effect/fetch.ts:64

---

state.FetchEffect.constructor

constructor

• **new FetchEffect**()

Public **constructor** (which should never be called).

`Throws`

A TypeError upon construction.

Inherited from Effect.constructor

Defined in packages/state/src/effect/effect.ts:71

---

state.FetchEffect.function

function

▸ **function**(`this`): (`requestInfo`: `URL` | `RequestInfo`, `requestInit?`: `RequestInit`) => `Promise<Response>`

Overridden **function** binding the FetchEffect to the polymorphic `this` of the StateWorker.

Parameters

Name	Type	Description
`this`	`StateWorker`	The explicit polymorphic `this` parameter.

Returns `fn`

This FetchEffect bound to the StateWorker.

▸ `(requestInfo, requestInit?): Promise<Response>`

Implanted FetchEffect providing convenient access to the globalThis.fetch method within Actions. Prefer the usage of this Effect over the globalThis.fetch method when dispatching Actions.

`Example`

Invoke **fetch** within an Action:

```
import { Stateful, Store } from '@sgrud/state';

@Stateful('io.github.sgrud.store.example', { response: undefined })
export class ExampleStore extends Store<ExampleStore> {

  public readonly response?: unknown;

  public async getResponse(url: string): Promise<Store.State<this>> {
    const request = await sgrud.state.effects.fetch(url);
    const response = await request.json();

    if (!request.ok) {
      throw response;
    }

    return { ...this, response };
  }

}
```

`See`

FetchEffect

Parameters

Name	Type	Description
`requestInfo`	`URL | RequestInfo`	The RequestInfo or URL to **fetch**.
`requestInit?`	`RequestInit`	An optional RequestInit object.

Returns `Promise<Response>`

A Promise of the **fetch**ed Response.

Overrides Effect.function

Defined in packages/state/src/effect/fetch.ts:73

---

state.Implant

Implant

▸ **Implant**<T, K>(locate): (constructor: T) => void

The **Implant** decorator, when applied to classes extending the abstract Effect base class, implants the decorated class under the `locate` in the global sgrud.state.effects namespace to be used within dispatched Actions.

Example

An `importScripts` **Effect**:

```
import { Effect, Implant, type StateWorker, type Store } from '@sgrud/state';

declare global {
  namespace sgrud.state.effects {
    function importScripts(...urls: (string | URL)[]): Promise<void>;
  }
}

@Implant('importScripts')
export class FetchEffect extends Effect {

  public override function(
    this: StateWorker
  ): Store.Effects['importScripts'] {
    return async(...urls) => {
      return importScripts(...urls);
    };
  }

}
```

See

- StateHandler
- Stateful

Type parameters

Name	Type	Description
T	extends () => Effect<K>	An Effect constructor type.
K	extends "fetch" | "state" | "dispatch"	The Effect `locate` type.

Parameters

Name	Type	Description
locate	K	The `locate` to address the Effect by.

Returns `fn`

A class constructor decorator.

▸ (constructor): void

Parameters

Name	Type
constructor	T

Returns `void`

Defined in packages/state/src/handler/implant.ts:45

---

state.IndexedDB

IndexedDB

• **IndexedDB**: `Object`

IndexedDB Driver. This class provides a facade derived from the built-in Storage interface to IDB-Databases within the browser. This class implementing the Driver contract is used as backing storage by the StateWorker, if run in a browser environment.

See

Driver

Defined in packages/state/src/driver/indexeddb.ts:11

---

state.IndexedDB.clear

clear

▸ **clear**(): `Promise<void>`

Removes all key/value pairs, if there are any.

Returns `Promise<void>`

A Promise resolving when this instance was **clear**ed.

Implementation of Store.Driver.clear

Defined in packages/state/src/driver/indexeddb.ts:69

---

state.IndexedDB.constructor

constructor

• **new IndexedDB**(`name`, `version`)

Public IndexedDB **constructor** consuming the `name` and `version` used to construct this instance of a Driver.

Parameters

Name	Type	Description
`name`	`string`	The `name` to address this instance by.
`version`	`string`	The `version` of this instance.

Defined in packages/state/src/driver/indexeddb.ts:38

---

state.IndexedDB.getItem

getItem

▸ **getItem**(`key`): `Promise<null | string>`

Returns the current value associated with the given `key`, or null if the given `key` does not exist.

Parameters

Name	Type	Description
`key`	`string`	The `key` to retrieve the current value for.

Returns `Promise<null | string>`

A Promise resolving to the current value or null.

Implementation of Store.Driver.getItem

Defined in packages/state/src/driver/indexeddb.ts:86

---

state.IndexedDB.key

key

▸ **key**(`index`): `Promise<null | string>`

Returns the name of the nth key, or null if n is greater than or equal to the number of key/value pairs.

Parameters

Name	Type	Description
`index`	`number`	The `index` of the **key** to retrieve.

Returns `Promise<null | string>`

A Promise resolving to the name of the **key** or null.

Implementation of Store.Driver.key

Defined in packages/state/src/driver/indexeddb.ts:103

---

state.IndexedDB.length

length

• `get` **length**(): `Promise<number>`

Returns the number of key/value pairs.

Returns `Promise<number>`

Implementation of Store.Driver.length

Defined in packages/state/src/driver/indexeddb.ts:21

---

state.IndexedDB.name

name

• `Readonly` **name**: `string`

The `name` to address this instance by.

Defined in packages/state/src/driver/indexeddb.ts:43

---

state.IndexedDB.removeItem

removeItem

▸ **removeItem**(key): `Promise<void>`

Removes the key/value pair with the given `key`, if a key/value pair with the given `key` exists.

Parameters

Name	Type	Description
`key`	`string`	The `key` to delete the key/value pair by.

Returns `Promise<void>`

A Promise resolving when the key/value pair was removed.

Implementation of Store.Driver.removeItem

Defined in packages/state/src/driver/indexeddb.ts:122

---

state.IndexedDB.setItem

setItem

▸ **setItem**(key, value): `Promise<void>`

Sets the `value` of the pair identified by `key` to `value`, creating a new key/value pair if none existed for `key` previously.

Parameters

Name	Type	Description
`key`	`string`	The `key` to set the key/value pair by.
`value`	`string`	The `value` to associate with the `key`.

Returns `Promise<void>`

A Promise resolving when the key/value pair was set.

Implementation of Store.Driver.setItem

Defined in packages/state/src/driver/indexeddb.ts:140

---

state.IndexedDB.version

version

• `Readonly` **version**: `string`

The `version` of this instance.

Defined in packages/state/src/driver/indexeddb.ts:48

---

state.IndexedDB.database

database

• `Private Readonly` **database**: `Promise<IDBDatabase>`

Private **database** used as backing storage to read/write key/value pairs.

Defined in packages/state/src/driver/indexeddb.ts:16

---

state.SQLite3

SQLite3

• **SQLite3**: `Object`

SQLite3 Driver. This class provides a facade derived from the built-in Storage interface to **SQLite3** databases under NodeJS. This class implementing the Driver contract is used as backing storage by the StateWorker, if run in a NodeJS environment.

See

Driver

Defined in packages/state/src/driver/sqlite3.ts:12

---

state.SQLite3.clear

clear

▸ **clear**(): `Promise<void>`

Removes all key/value pairs, if there are any.

Returns `Promise<void>`

A Promise resolving when this instance was **clear**ed.

Implementation of Store.Driver.clear

Defined in packages/state/src/driver/sqlite3.ts:76

---

state.SQLite3.constructor

constructor

• **new SQLite3**(`name`, `version`)

Public SQLite3 **constructor** consuming the `name` and `version` used to construct this instance of a Driver.

Parameters

Name	Type	Description
`name`	`string`	The `name` to address this instance by.
`version`	`string`	The `version` of this instance.

Defined in packages/state/src/driver/sqlite3.ts:39

---

state.SQLite3.getItem

getItem

▸ **getItem**(key): `Promise<null | string>`

Returns the current value associated with the given `key`, or null if the given `key` does not exist.

Parameters

Name	Type	Description
`key`	`string`	The `key` to retrieve the current value for.

Returns `Promise<null | string>`

A Promise resolving to the current value or null.

Implementation of Store.Driver.getItem

Defined in packages/state/src/driver/sqlite3.ts:93

---

state.SQLite3.key

key

▸ **key**(index): `Promise<null | string>`

Returns the name of the nth key, or null if n is greater than or equal to the number of key/value pairs.

Parameters

Name	Type	Description
`index`	`number`	The `index` of the **key** to retrieve.

Returns `Promise<null | string>`

A Promise resolving to the name of the **key** or null.

Implementation of Store.Driver.key

Defined in packages/state/src/driver/sqlite3.ts:110

---

state.SQLite3.length

length

• `get` **length**(): `Promise<number>`

Returns the number of key/value pairs.

Returns `Promise<number>`

Implementation of Store.Driver.length

Defined in packages/state/src/driver/sqlite3.ts:22

---

state.SQLite3.name

name

• `Readonly` **name**: `string`

The `name` to address this instance by.

Defined in packages/state/src/driver/sqlite3.ts:44

---

state.SQLite3.removeItem

removeItem

▸ **removeItem**(`key`): `Promise<void>`

Removes the key/value pair with the given `key`, if a key/value pair with the given `key` exists.

Parameters

Name	Type	Description
`key`	`string`	The `key` to delete the key/value pair by.

Returns `Promise<void>`

A Promise resolving when the key/value pair was removed.

Implementation of Store.Driver.removeItem

Defined in packages/state/src/driver/sqlite3.ts:127

---

state.SQLite3.setItem

setItem

▸ **setItem**(`key`, `value`): `Promise<void>`

Sets the `value` of the pair identified by `key` to `value`, creating a new key/value pair if none existed for `key` previously.

Parameters

Name	Type	Description
`key`	`string`	The `key` to set the key/value pair by.
`value`	`string`	The `value` to associate with the `key`.

Returns `Promise<void>`

A Promise resolving when the key/value pair was set.

Implementation of Store.Driver.setItem

Defined in packages/state/src/driver/sqlite3.ts:145

---

state.SQLite3.version

version

• `Readonly` **version**: `string`

The `version` of this instance.

Defined in packages/state/src/driver/sqlite3.ts:49

---

state.SQLite3.database

database

• `Private` `Readonly` **database**: `Database`

Private **database** used as backing storage to read/write key/value pairs.

Defined in packages/state/src/driver/sqlite3.ts:17

---

state.StateEffect

StateEffect

• **StateEffect**: `Object`

Built-in **StateEffect** extending the abstract Effect base class. This **StateEffect** is automatically implanted when the `@sgrud/state` module is imported and can therefore be always used in Actions.

`Decorator`

Implant

`See`

Effect

Defined in packages/state/src/effect/state.ts:61

---

state.StateEffect.constructor

constructor

• **new StateEffect**()

Public **constructor** (which should never be called).

`Throws`

A TypeError upon construction.

Inherited from Effect.constructor

Defined in packages/state/src/effect/effect.ts:71

---

state.StateEffect.function

function

▸ **function**(`this`): <T>(`handle`: `` `${string}.${string}.${string}` ``) => `Promise<State<T> | undefined>`

Overridden **function** binding the StateEffect to the polymorphic `this` of the StateWorker.

Parameters

Name	Type	Description
`this`	`StateWorker`	The explicit polymorphic `this` parameter.

Returns `fn`

This StateEffect bound to the StateWorker.

▸ <T>(handle): `Promise<State<T> | undefined>`

Implanted StateEffect providing convenient access to the **state** of Stores from within dispatched Actions. This StateEffect in combination with the DispatchEffect can be used to implement complex interactions between different Stores.

`Example`

Retrieve the **state** from another Store:

```
import { type Bus } from '@sgrud/bus';
import { Stateful, Store } from '@sgrud/state';

@Stateful('io.github.sgrud.store.example', { state: undefined })
export class ExampleStore extends Store<ExampleStore> {

  public readonly state?: Store.State<Store>;

  public async getState(handle: Bus.Handle): Promise<Store.State<this>> {
    const state = await sgrud.state.effects.state(handle);

    return { ...this, state };
  }

}
```

`See`

StateEffect

Type parameters

Name	Type	Description
T	extends `Store<any, T>`	The extending Store InstanceType.

Parameters

Name	Type	Description
handle	'${string}.${string}.${string}'	The Handle representing the Store.

Returns `Promise<State<T> | undefined>`

The current **state** of the Store.

Overrides Effect.function

Defined in packages/state/src/effect/state.ts:70

---

state.StateHandler

StateHandler

• **StateHandler**: `Object`

The **StateHandler** Singleton class provides the means to interact with an automatically registered ServiceWorker, when instantiated in a browser environment or, when the **StateHandler** is instantiated within a NodeJS environment, a `new require('worker_threads').Worker()` is forked. Within either of these Threads the StateWorker is executed and handles the deployment of Stores and dispatching Actions against them. The same goes for Effects, whose implantation the StateWorker handles.

The functionality provided by the **StateHandler** is best consumed by applying on of the Stateful or Implant decorators, as those provide easier and higher-level interfaces to the functionality provided by this Singleton class.

`Decorator`

Singleton

`See`

StateWorker

Defined in packages/state/src/handler/handler.ts:30

---

state.StateHandler.[observable]

[observable][]

▸ `Static` **[observable]**(): `Subscribable<StateHandler>`

Static `Symbol.observable` method returning a Subscribable. The returned Subscribable mirrors the private loader and is used for initializations after the StateHandler has been successfully initialized.

`Example`

Subscribe to the StateHandler:

```
import { StateHandler } from '@sgrud/state';
import { from } from 'rxjs';

from(StateHandler).subscribe(console.log);
```

Returns `Subscribable<StateHandler>`

A Subscribable emitting this StateHandler.

Defined in packages/state/src/handler/handler.ts:56

---

state.StateHandler.loader

loader

■ `Static` `Private` **loader**: `ReplaySubject<StateHandler>`

Private static ReplaySubject used as the StateHandler **loader**. This **loader** emits once after the StateHandler has been successfully initialized.

Defined in packages/state/src/handler/handler.ts:37

---

state.StateHandler.constructor

constructor

• **new StateHandler**(`source?`, `scope?`)

Public StateHandler **constructor**. As the StateHandler is a Singleton class, this **constructor** is only invoked the first time it is targeted by the `new` operator. Upon this first invocation, the worker property is assigned an instance of the StateWorker Thread while using the supplied `source`, if any.

`Throws`

A ReferenceError when the environment is incompatible.

Parameters

Name	Type	Description
`source?`	`string`	An optional Module `source`.
`scope?`	`string`	An optionally `scoped` ServiceWorkerRegistration.

Defined in packages/state/src/handler/handler.ts:95

---

state.StateHandler.deploy

deploy

▸ **deploy**<T>(`handle`, `store`, `state`, `transient?`): `Observable<void>`

Public **deploy** method which defers the **deploy**ment of the supplied `store` under the supplied `handle` to the StateWorker. For convenience, instead of invoking this **deploy** method manually, the Stateful decorator should be considered.

Type parameters

Name	Type	Description
`T`	extends `Store<any, T>`	The extending Store InstanceType.

Parameters

Name	Type	Default value	Description
`handle`	‘${string}.${string}.${string}‘	`undefined`	The Handle representing the Store.
`store`	`Type<T>`	`undefined`	The Store to **deploy** under the supplied `handle`.
`state`	`State<T>`	`undefined`	An initial State for the Store.
`transient`	`boolean`	`false`	Wether the Store is considered `transient`.

Returns `Observable<void>`

An Observable of the Store **deploy**ment.

Defined in packages/state/src/handler/handler.ts:155

---

state.StateHandler.deprecate

deprecate

▸ **deprecate**(`handle`): `Observable<void>`

Public **deprecate** method which defers to an invocation of the backing **deprecate** method of the StateWorker to **deprecate** the Store represented by the supplied `handle`.

Parameters

Name	Type	Description
`handle`	‘${string}.${string}.${string}‘	The Handle representing the Store.

Returns `Observable<void>`

An Observable of the Store deprecation.

Defined in packages/state/src/handler/handler.ts:174

---

state.StateHandler.dispatch

dispatch

▸ **dispatch**<T>(handle, ...action): Observable<State<T>>

Public **dispatch** method which defers the **dispatch**ing of the supplied `action` to the Store represented by the the supplied `handle` to the StateWorker. For convenience, instead of manually invoking this **dispatch** method manually, the Stateful decorator should be considered.

Type parameters

Name	Type	Description
T	extends Store<any, T>	The extending Store InstanceType.

Parameters

Name	Type	Description
handle	‘${string}.${string}.${string}‘	The Handle representing the Store.
...action	Action<T>	A type-guarded Action to **dispatch**.

Returns `Observable<State<T>>`

An Observable of the resulting State.

Defined in packages/state/src/handler/handler.ts:192

---

state.StateHandler.implant

implant

▸ **implant**<K>(locate, effect): Observable<void>

Public **implant** method which defers the **implant**ation of the supplied `effect` under the supplied `locate` to the StateWorker. For convenience, instead of invoking this **implant** method manually, the Implant decorator should be considered.

Type parameters

Name	Type	Description
K	extends "fetch"	"state"	"dispatch"	The Effect `locate` type.

Name	Type	Description

Parameters

Name	Type	Description
`locate`	`K`	The `locate` to address the Effect by.
`effect`	`() => Effect<K>`	The Effect to **implant** under the `locate`.

Returns `Observable<void>`

An Observable of the Store **implant**ation.

Defined in packages/state/src/handler/handler.ts:212

---

state.StateHandler.invalidate

invalidate

▸ **invalidate**<K>(`locate`): `Observable<void>`

Public **invalidate** method which defers to an invocation of the backing **invalidate** method of the StateWorker to **invalidate** the Effect represented by the supplied `locate`.

Type parameters

Name	Type	Description
`K`	extends `"fetch"` | `"state"` | `"dispatch"`	The Effect `locate` type.

Parameters

Name	Type	Description
`locate`	`K`	The `locate` to address the Effect by.

Returns `Observable<void>`

An Observable of the Effect invalidation.

Defined in packages/state/src/handler/handler.ts:230

---

state.StateHandler.worker

worker

• `Readonly` **worker**: `Thread<StateWorker>`

The **worker** Thread is the main background workhorse, depending on the environment, either a `navigator.serviceWorker` is `registered` or a `new require('worker_threads').Worker()` NodeJS equivalent will be forked.

See

StateWorker

Defined in packages/state/src/handler/handler.ts:74

---

state.StateHandler.kernel

kernel

• `Private` `Readonly` **kernel**: `Kernel`

Factored-in **kernel** property linking the Kernel.

`Decorator`

Factor

Defined in packages/state/src/handler/handler.ts:82

---

state.StateWorker

StateWorker

• **StateWorker**: `Object`

The **StateWorker** is a background Thread which is instantiated by the StateHandler to handle the deployment of Stores and dispatching Actions against them. The same goes for Effects, whose implantation the StateWorker handles.

`Decorator`

Singleton

`See`

StateHandler

Defined in packages/state/src/worker/index.ts:35

---

state.StateWorker.activate

activate

▸ `Static` `Private` **activate**(`event`): `void`

Private static **activate** method, called when this StateWorker is instantiated as ServiceWorker in a browser environment upon activation of the ServiceWorker.

Parameters

Name	Type	Description
`event`	`ExtendableEvent`	The fired ExtendableEvent.

Returns `void`

Defined in packages/state/src/worker/index.ts:70

---

state.StateWorker.install

install

▸ `Static` `Private` **install**(`event`): `void`

Private static **install** method, called when this StateWorker is instantiated as ServiceWorker in a browser environment upon installation of the ServiceWorker.

Parameters

Name	Type	Description
`event`	`ExtendableEvent`	The fired ExtendableEvent.

Returns `void`

Defined in packages/state/src/worker/index.ts:81

---

state.StateWorker.message

message

▸ `Static` `Private` **message**(`event`): `void`

Private static **message** method, called when this StateWorker is instantiated as ServiceWorker in a browser environment upon the reception of messages from the controlling Window.

Parameters

Name	Type	Description
`event`	`ExtendableMessageEvent`	The fired ExtendableMessageEvent.

Returns `void`

Defined in packages/state/src/worker/index.ts:92

---

state.StateWorker.connect

connect

▸ **connect**(`socket`): `Promise<void>`

Public **connect** method which **connect**s this StateWorker to a BusWorker through the supplied `socket`.

`Remarks`

This method should only be invoked by the StateHandler.

Parameters

Name	Type	Description
`socket`	`MessagePort`	A MessagePort to the BusWorker.

Returns `Promise<void>`

A Promise resolving upon `socket` **connect**ion.

Defined in packages/state/src/worker/index.ts:179

---

state.StateWorker.constructor

constructor

• **new StateWorker**(`source`)

Public Singleton StateWorker **constructor**. As this is a Singleton **constructor** it is only invoked the first time this StateWorker class is targeted by the `new` operator. Furthermore this **constructor** returns, depending of the presence of the `source` parameter, a proxyfied instance of this StateWorker class instead of the actual `this` reference.

`Remarks`

This method should only be invoked by the StateHandler.

Parameters

Name	Type	Description
`source`	`null` | `MessagePort` | `Client` | `ServiceWorker`	The initial ExtendableMessageEvent `source`.

Defined in

packages/state/src/worker/index.ts:156

---

state.StateWorker.deploy

deploy

▸ **deploy**<`T`>(`handle`, `store`, `state`, `transient?`): `Promise<void>`

Public **deploy** method which **deploy**s the supplied `store` under the supplied `handle`. If the Store is **deploy**ed `transiently`, the supplied `state` is used as initial State. Otherwise, if a previously persisted State exists, it takes precedence over the supplied `state`. Furthermore, when the supplied Type is already **deploy**ed and matches the currently **deploy**ed source code, no action is taken. If the `store`'s sources mismatch, a TypeError is thrown.

`Throws`

A TypeError when the supplied `store` mismatches.

`Remarks`

This method should only be invoked by the StateHandler.

Type parameters

Name	Type	Description
`T`	extends `Store<any, T>`	The extending Store InstanceType.

Parameters

Name	Type	Default value	Description
`handle`	`'${string}.${string}.${string}'`	`undefined`	The Handle representing the Store.
`store`	`Type<T>`	`undefined`	The Store to **deploy** under the supplied `handle`.
`state`	`State<T>`	`undefined`	An initial State for the Store.
`transient`	`boolean`	`false`	Wether the Store is considered `transient`.

Returns

`Promise<void>`

A Promise resolving upon Store **deploy**ment.

Defined in packages/state/src/worker/index.ts:203

---

state.StateWorker.deprecate

deprecate

▸ **deprecate**(`handle`): `Promise<void>`

Public **deprecate** method. When the returned Promise resolves, the deployed Store referenced by the supplied `handle` is guaranteed to be **deprecate**d. Otherwise a ReferenceError is thrown (and therefore the returned Promise rejected).

`Throws`

A ReferenceError when no Store could be `handle`d.

`Remarks`

This method should only be invoked by the StateHandler.

Parameters

Name	Type	Description
`handle`	`'${string}.${string}.${string}'`	The Handle representing the Store.

Returns `Promise<void>`

A Promise resolving upon Store deprecation.

Defined in packages/state/src/worker/index.ts:278

---

state.StateWorker.dispatch

dispatch

▸ **dispatch**<`T`>(`handle`, `action`): `Promise<State<T>>`

Public **dispatch** method. Invoking this method while supplying a `handle` and a appropriate `action` will apply the supplied Action against the Store deployed under the supplied `handle`. The returned Promise resolves to the resulting new State of the Store after the supplied Action was **dispatch**ed against it.

`Throws`

A ReferenceError when no Store could be `handle`d.

`Remarks`

This method should only be invoked by the StateHandler.

Type parameters

Name	Type	Description
`T`	extends `Store<any, T>`	The extending Store InstanceType.

Parameters

Name	Type	Description
`handle`	`'${string}.${string}.${string}'`	The Handle representing the Store.
`action`	`Action<T>`	A type-guarded Action to **dispatch**.

Returns `Promise<State<T>>`

A Promise resolving to the resulting State.

Defined in packages/state/src/worker/index.ts:308

---

state.StateWorker.implant

implant

▸ **implant**<K>(`locate`, `effect`): `Promise<void>`

Public **implant** method which **implant**s the supplied `effect` under the supplied `locate` to the global sgrud.state.effects namespace. When the supplied Effect is already **implant**ed and matches the currently **implant**ed source code, no action is taken. If the `effect`'s sources mismatch, a TypeError is thrown.

`Throws`

A TypeError when the supplied `effect` mismatches.

`Remarks`

This method should only be invoked by the StateHandler.

Type parameters

Name	Type	Description
`K`	extends `"fetch"` | `"state"` | `"dispatch"`	The Effect `locate` type.

Parameters

Name	Type	Description
`locate`	`K`	The `locate` to address the Effect by.
`effect`	() => `Effect<K>`	The Effect to **implant** under the `locate`.

Returns `Promise<void>`

A Promise resolving upon Store **implant**ation.

Defined in packages/state/src/worker/index.ts:343

---

state.StateWorker.invalidate

invalidate

▸ **invalidate**<K>(`locate`): `Promise<void>`

Public **invalidate** method. When the returned Promise resolves, the implanted Effect referenced by the supplied `locale` is guaranteed to be **invalidate**d. Otherwise a ReferenceError is thrown (and therefore the returned Promise rejected).

`Throws`

A ReferenceError when no Effect could be `locate`d.

`Remarks`

This method should only be invoked by the StateHandler.

Type parameters

Name	Type	Description
K	extends "fetch" | "state" | "dispatch"	The Effect locate type.

Parameters

Name	Type	Description
locate	K	The locate to address the Effect by.

Returns Promise<void>

A Promise resolving upon Effect invalidation.

Defined in packages/state/src/worker/index.ts:369

---

state.StateWorker.driver

driver

• Protected Readonly **driver**: Driver

Internal Driver employed as backing data storage. This property contains an instance of either the IndexedDB or the SQLite3 class as abstract facade to either storage provider.

Defined in packages/state/src/worker/index.ts:104

---

state.StateWorker.effects

effects

• Protected Readonly **effects**: Map<"fetch" | "state" | "dispatch", Function>

Internal Mapping of Effect locates to their corresponding bound Effects.

Defined in packages/state/src/worker/index.ts:110

---

state.StateWorker.proxies

proxies

• Protected Readonly **proxies**: WeakMap<object, typeof effects>

Internal WeakMapping of proxyfied references to this StateWorker to the Effects namespace containing Effects bound to this StateWorker.

Defined in packages/state/src/worker/index.ts:117

---

state.StateWorker.remotes

remotes

• Protected Readonly **remotes**: Map<StateWorker, Remote<BusWorker>>

Internal Mapping of Remote BusWorkers to their corresponding proxy of this StateWorker. This Map is used to keep track of the connected Windows and their respective BusWorkers.

Defined in packages/state/src/worker/index.ts:125

---

state.StateWorker.states

states

• `Protected` `Readonly` **states**: `Map<'${string}.${string}.${string}', WeakMap<object, States>>`

Internal Mapping of Handles to WeakMapping of States designated by an object reference. This reference either points to the global self reference, if a Store is deployed to be non-transient or, if the opposite applies, to the proxyfied instance of this StateWorker. Through this distinction stores are associated to either a globally shared reference or to a locally contained and transparent Proxy reference to `this`.

Defined in packages/state/src/worker/index.ts:136

---

state.StateWorker.stores

stores

• `Protected` `Readonly` **stores**: `Map<'${string}.${string}.${string}', Type<Store<any>>>`

Internal Mapping of deployed Types to their corresponding Handles.

Defined in packages/state/src/worker/index.ts:142

---

state.StateWorker.proxy

proxy

▸ `Private` **proxy**(`source`): `StateWorker`

Private **proxy** method wrapping this StateWorker instance in a Proxy. The resulting Proxy is used to provide distinct `this` references for each of the connected remotes and intercepts dispatch invocations to provide the globally available sgrud.state.effects namespace.

Parameters

Name	Type	Description
`source`	`MessagePort | Client | ServiceWorker`	The initial ExtendableMessageEvent `source`.

Returns `StateWorker`

A Proxy wrapping the StateWorker.

Defined in packages/state/src/worker/index.ts:385

---

state.Stateful

Stateful

▸ **Stateful**<`T`, `I`>(`handle`, `state`, `transient?`): (`constructor`: `T`) => `T`

The **Stateful** decorator, when applied to classes extending the abstract Store base class, converts those extending classes into type-guarding Store facades implementing only the dispatch and the well-known `Symbol.observable` methods. This resulting facade provides convenient access to the current and upcoming States of the decorated Store and its dispatch method. The decorated class is deployed under the supplied `handle` using the supplied `state` as an initial State. If the Store is to be deployed `transient`ly, the supplied `state` is guaranteed to be used as initial State. Otherwise, a previously persisted State takes precedence over the supplied `state`.

Example

A simple `ExampleStore` facade:

```ts
import { Stateful, Store } from '@sgrud/state';

@Stateful('io.github.sgrud.store.example', {
  property: 'default',
  timestamp: Date.now()
})
export class ExampleStore extends Store<ExampleStore> {

  public readonly property!: string;

  public readonly timestamp!: number;

  public async action(property: string): Promise<Store.State<this>> {
    return { ...this, property, timestamp: Date.now() };
  }

}
```

Example

Subscribe to the `ExampleStore` facade:

```ts
import { ExampleStore } from './example-store';

const store = new ExampleStore();
from(store).subscribe(console.log);
// { property: 'default', timestamp: [...] }
```

Example

Dispatch an Action through the `ExampleStore` facade:

```ts
import { ExampleStore } from './example-store';

const store = new ExampleStore();
store.dispatch('action', ['value']).subscribe(console.log);
// { property: 'value', timestamp: [...] }
```

See

- StateHandler
- Implant

Type parameters

Name	Type	Description
`T`	extends `Type<I, T>`	A constructor type extending the Type.
`I`	extends `Store<any, I>` = `InstanceType<T>`	The extending Store InstanceType.

Parameters

Name	Type	Default value	Description
`handle`	`'${string}.${string}.${string}'`	`undefined`	The Handle representing the Store.
`state`	`State<I>`	`undefined`	An initial State for the Store.
`transient`	`boolean`	`false`	Wether the Store is considered `transient`.

Returns `fn`

A class constructor decorator.

▸ `(constructor): T`

Parameters

Name	Type
`constructor`	`T`

Returns `T`

Defined in packages/state/src/handler/stateful.ts:71

---

state.Store

Store

• `Abstract` **Store**<`T`>: `Object`

Abstract **Store** base class. By extending this **Store** base class and decorating the extending class with the Stateful decorator, the resulting **Store** will become a functional facade implementing only the dispatch and well-known `Symbol.observable` methods. This resulting facade provides convenient access to the current and upcoming States of the **Store** and its dispatch method, while, behind the facade, interactions with the BusHandler to provide an Observable of the State changes and the StateHandler to dispatch any Actions will be handled transparently.

The same functionality can be achieved by manually supplying a **Store** to the StateHandler and subscribing to the changes of that **Store** through the BusHandler while any Actions also have to be passed manually to the StateHandler. But the Stateful decorator should be preferred out of convenience and because invoking the constructor of the **Store** class throws a TypeError.

Example

A simple `ExampleStore` facade:

```
import { Stateful, Store } from '@sgrud/state';

@Stateful('io.github.sgrud.store.example', {
  property: 'default',
  timestamp: Date.now()
})
export class ExampleStore extends Store<ExampleStore> {

  public readonly property!: string;

  public readonly timestamp!: number;

  public async action(property: string): Promise<Store.State<this>> {
    return { ...this, property, timestamp: Date.now() };
  }

}
```

Example

Subscribe to the `ExampleStore` facade:

```
import { ExampleStore } from './example-store';

const store = new ExampleStore();
from(store).subscribe(console.log);
// { property: 'default', timestamp: [...] }
```

Example

Dispatch an Action through the `ExampleStore` facade:

```
import { ExampleStore } from './example-store';

const store = new ExampleStore();
store.dispatch('action', ['value']).subscribe(console.log);
// { property: 'value', timestamp: [...] }
```

Type parameters

Name	Type	Description
T	extends Store = any	The extending Store InstanceType.

Defined in packages/state/src/store/store.ts:11

packages/state/src/store/store.ts:163

---

state.Store.[observable]

[observable][]

• **[observable]**: () => Subscribable<State<T>>

Type declaration ▸ (): Subscribable<State<T>>

Well-known Symbol.observable method returning a Subscribable. The returned Subscribable emits all States this Store traverses, i.e., all States that result from dispatching Actions on this Store.

Throws

An ReferenceError when not called Stateful.

Example

Subscribe to the ExampleStore:

```
import { ExampleStore } from './example-store';

const store = new ExampleStore();
from(store).subscribe(console.log);
```

Returns Subscribable<State<T>>

A Subscribable emitting State changes.

Defined in packages/state/src/store/store.ts:183

---

state.Store.constructor

constructor

• **new Store**<T>()

Throws

A TypeError upon construction.

Type parameters

Name	Type
T	extends Store<any, T> = any

Defined in packages/state/src/store/store.ts:188

---

state.Store.dispatch

dispatch

▸ **dispatch**(...action): `Observable<State<T>>`

The **dispatch** method provides a facade to **dispatch** an Action through the StateHandler when this Store was decorated with the Stateful decorator, otherwise calling this method will throw an ReferenceError.

`Throws`

An ReferenceError when not called Stateful.

`Example`

Dispatch an Action to the `ExampleStore`:

```
import { ExampleStore } from './example-store';

const store = new ExampleStore();
store.dispatch('action', ['value']).subscribe();
```

Parameters

Name	Type	Description
...action	`Action<T>`	A type-guarded Action to **dispatch**.

Returns `Observable<State<T>>`

An Observable of the resulting State.

Defined in packages/state/src/store/store.ts:211

---

state.Store

Store

• **Store**: `Object`

The **Store** namespace contains types and interfaces used and intended to be used in conjunction with the abstract Store class.

`See`

Store

Defined in packages/state/src/store/store.ts:11

packages/state/src/store/store.ts:163

---

state.Store.Action

Action

T **Action**<T>: { [K in Exclude<keyof T, keyof Store<T>>]: T[K] extends Function ? [K] : T[K] extends Function ? [K, I] : never }[`Exclude<keyof T, keyof Store<T>>`]

This Store **Action** helper type represents the signatures of all available **Action**s of any given Store by extracting all methods from the given Store that return a promisified State of that given Store. This State is interpreted as the next State after this **Action** was invoked.

Type parameters

Name	Type	Description
T	extends `Store`	The extending Store InstanceType.

Defined in packages/state/src/store/store.ts:23

---

state.Store.Driver

Driver

T **Driver**: { [K in keyof Storage as string extends K ? never : K]: Storage[K] extends Function ? Function : Promise<Storage[K]> }

The **Driver** helper type is a promisified variant of the built-in Storage type. This type is utilized by the StateWorker where it represents one of the available Storage **Driver**s.

Defined in packages/state/src/store/store.ts:39

---

state.Store.Effect

Effect

T **Effect**: keyof `Effects`

The **Effect** helper type represents a `keyof` the Effects map.

Defined in packages/state/src/store/store.ts:49

---

state.Store.Effects

Effects

T **Effects**: typeof `sgrud.state.effects`

The **Effects** helper type represents the `typeof` the globally available sgrud.state.effects namespace.

Defined in packages/state/src/store/store.ts:55

---

state.Store.State

State

T **State**<T>: { readonly [P in { [K in Exclude<keyof T, keyof Store<T>>]: T[K] extends Function ? never : K }[Exclude<keyof T, keyof Store<T>>]]: T[P] }

The Store **State** helper type represents the current **State** of any given Store by extracting all properties (and dropping any methods) from that given Store.

Type parameters

Name	Type	Description
`T`	extends `Store`	The extending Store InstanceType.

Defined in packages/state/src/store/store.ts:64

---

state.Store.States

States

T **States**: `BehaviorSubject<State<Store>>`

The **States** helper type represents the traversal of Stores.

Defined in packages/state/src/store/store.ts:76

---

state.Store.Type

Type

• **Type**<T>: `Object`

Interface describing the **Type**, i.e., static constructable context, of classes extending the abstract Store base class.

Type parameters

Name	Type	Description
T	extends Store	The extending Store InstanceType.

Defined in packages/state/src/store/store.ts:84

Store.Type.constructor

constructor

• **new Type**()

Overridden and concretized constructor signature.

Inherited from Required<typeof Store>.constructor

Defined in packages/state/src/store/store.ts:94

Store.Type.prototype

prototype

• `Readonly` **prototype**: T

Overridden `prototype` signature.

Overrides Required.prototype

Defined in packages/state/src/store/store.ts:89

Index